

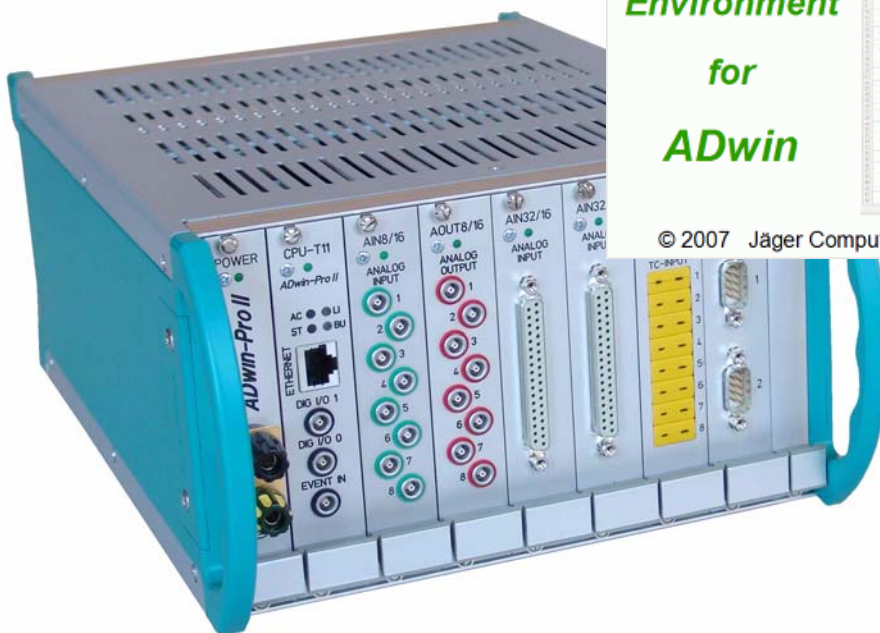


**JÄGER**

Computergesteuerte  
Messtechnik GmbH

# ***ADwin-Pro II***

## **System specifications Programming in *ADbasic***

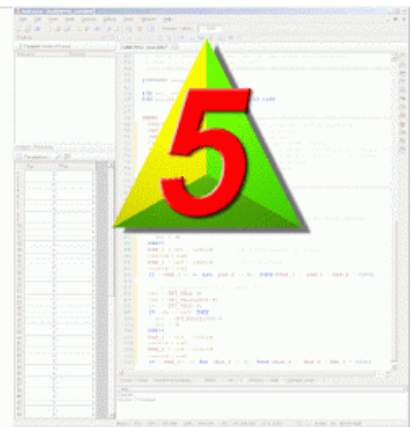


***ADbasic***

*Real-Time  
Development  
Environment*

*for*

***ADwin***



© 2007 Jäger Computergesteuerte Messtechnik GmbH

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

Hotline: +49 6251 96320  
Fax: +49 6251 568 19  
E-Mail: [info@ADwin.de](mailto:info@ADwin.de)  
Internet: [www.ADwin.de](http://www.ADwin.de)



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

## Table of contents

Table of contents .....	III
Typographical Conventions .....	IV
1 Introduction .....	1
2 The Program ADpro.exe .....	2
3 <i>ADbasic</i> instruction for <i>ADwin-Pro II</i> modules .....	3
3.1 Pro II: All Modules .....	4
3.2 Pro II: Multi-I/O .....	24
3.3 Pro II: Analog Input Modules .....	33
3.4 Pro II: Output Modules .....	116
3.5 Pro II: Digital I/O Modules .....	131
3.6 Pro II: Counter Modules .....	161
3.7 Pro II: PWM Output Modules .....	188
3.8 Pro II: Temperature Measuring Modules .....	199
3.9 Pro II: CAN bus Modules .....	217
3.10 Pro II: RSxxx Modules .....	239
3.11 Pro II: LIN bus Interface .....	250
3.12 Pro II: Profibus interface .....	263
3.13 Pro II: EtherCAT interface .....	267
3.14 Pro II: FlexRay .....	274
4 Program Examples .....	283
4.1 Continuous signal conversion .....	283
Instruction Lists .....	A-1
A.1 Alphabetic Instruction List .....	A-1
A.2 Instruction List sorted by Module Types .....	A-4
A.3 Thematic Instruction List .....	A-13

## Typographical Conventions



"Warning" stands for information, which indicate damages of hardware or software, test setup or injury to persons caused by incorrect handling.



You find a "note" next to

- information, which absolutely have to be considered in order to guarantee an error free operation.
- advice for efficient operation.



"Information" refers to further information in this documentation or to other sources such as manuals, data sheets, literature, etc.

<C:\ADwin\ ...>

File names and paths are placed in <angle brackets> and characterized in the font *Courier New*.

Program text

Program instructions and user inputs are characterized by the font *Courier New*.

Var\_1

ADbasic source code elements such as instructions, variables, comments and other text are characterized by the font *Courier New* and are printed in color (see also the editor of the *ADbasic* development environment).

Bits in data (here: 16 bit) are referred to as follows:

Bit No.	15	14	13	...	01	00
Bit value	$2^{15}$	$2^{14}$	$2^{13}$	...	$2^1=2$	$2^0=1$
Synonym	MSB	-	-	-	-	LSB

### 1 Introduction

The real-time development tool *ADbasic* is a software that on the one hand is a means for easy programming of the *ADwin-Pro II* processor system and on the other hand is a tool that completely uses the multi-processing capacities of the system.

This manual describes *ADbasic* instructions to access the variety of modules. (see annex for the [Instruction List sorted by Module Types](#)).

There is also the *ADbasic* manual which describes the more basic command e.g. for calculations, for program structure or for process control.

The commands for access to the *ADwin-Pro II* system with *ADbasic* are included in the include files. After installation from the *ADwin* CD-ROM the include files are available in the directory <C:\ADwin\ADbasic\inc>.

In order to get access to the *ADwin-Pro* and *ADwin-Pro II* modules you include all required include files with the following line into your *ADbasic* program.

```
#INCLUDE ADwinPRO_ALL.inc
```

If you have already written *ADbasic* programs, you have used a separate include file for each module group. Delete all these include lines completely and insert the upper line instead.



#### Please note:

For *ADwin* systems to function correctly, adhere strictly to the information provided in this documentation and in other mentioned manuals.

Programming, start-up and operation, as well as the modification of program parameters must be performed only by appropriately qualified personnel.

*Qualified personnel are persons who, due to their education, experience and training as well as their knowledge of applicable technical standards, guidelines, accident prevention regulations and operating conditions, have been authorized by a quality assurance representative at the site to perform the necessary activities, while recognizing and avoiding any possible dangers.*

*(Definition of qualified personnel as per VDE 105 and ICE 364).*

This product documentation and all documents referred to, have always to be available and to be strictly observed. For damages caused by disregarding the information in this documentation or in all other additional documentations, no liability is assumed by the company *Jäger Computergesteuerte Messtechnik GmbH*, Lorsch, Germany.

This documentation, including all pictures is protected by copyright. Reproduction, translation as well as electronical and photographic archiving and modification require a written permission by the company *Jäger Computergesteuerte Messtechnik GmbH*, Lorsch, Germany.

OEM products are mentioned without referring to possible patent rights, the existence of which, may not be excluded.

Hotline address: see inner side of cover page.

#### Qualified personnel

#### Availability of the documents



#### Legal information

#### Subject to change.

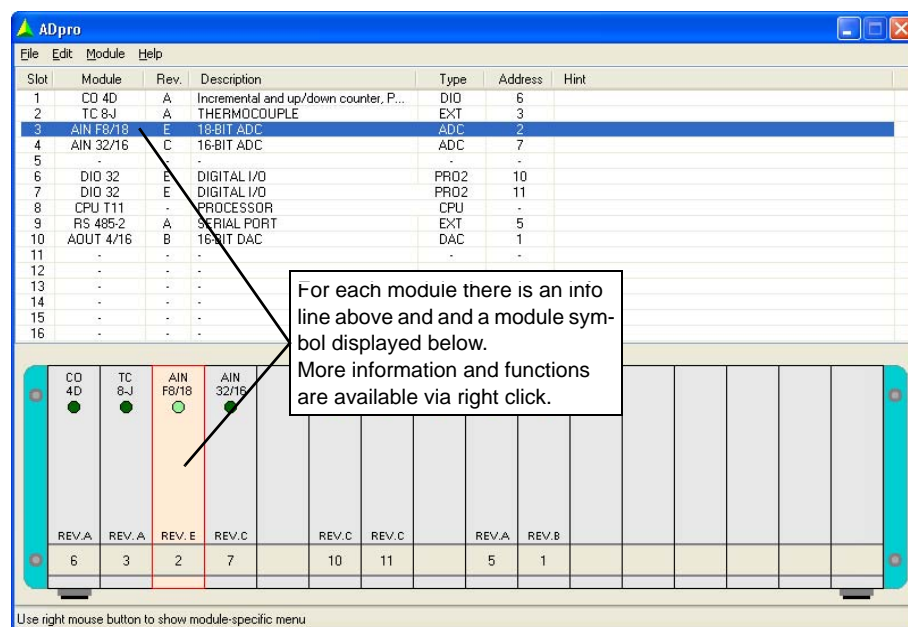
## 2 The Program ADpro.exe

The program tool <ADpro.exe> has several tasks:

- Show the modules on an *ADwin-Pro II* system as well as information on the modules.
- Set the module address for Pro II modules (see hardware manual).  
With Pro I modules the module address is set manually; here the address can be shown only.
- Check the function of Pro I and Pro II modules: analog input / output modules, digital and counter modules, some bus modules.
- Calibrate Pro I and Pro II modules (analog input / output modules).

The calibration meets lower demands only.

The use of the program ADpro is self-explanatory; some functions are available via context menu (right mouse click). Please note the accompanying text and follow the hints given.



### Notes

ADpro.exe initializes the *ADwin* system, thus ending and deleting still running processes.

If there is an error upon start-up of the program, please check if the software packet <Microsoft .NET Framework 2.0> is installed on the PC.

### 3 ADbasic instruction for ADwin-Pro II modules

This section contains all instructions to access *ADwin-Pro II* modules. The instructions are sorted according to module groups and then alphabetically.

In the annex you find furthermore the following sorted instruction lists:

- [Alphabetic Instruction List](#)
- [Instruction List sorted by Module Types](#)

Use the module's list of valid instructions to learn about the functions of a module.

- [Thematic Instruction List](#)

Instructions for *ADwin-Pro I* and *ADwin-Pro II* modules often are quite similar. For distinction, Pro II instructions have the prefix **P2\_**.

To use an instruction you have to include the following line into your *ADbasic* program:

```
#Include ADwinPro_All.Inc
```

The description for each instruction includes:

- syntax and passed parameter.
- notes about specific features.
- a list of related instructions.
- a list of modules where the instruction is applicable.
- often an example.

The examples (mostly) assume the module address to be set to the number 1.

All Pro II modules, which are accessed via an *ADbasic* instruction, must be plugged-in correctly. Otherwise the processor workload rises, even the communication to the PC may be interrupted.

Unlike the Pro I modules an access attempt to a non-accessible Pro II module lasts longer than if the module is accessible. This may happen for example, when a Pro II module is unplugged. The elongated access time increases the workload of the CPU module and changes the process timing.



### 3.1 Pro II: All Modules

This section describes instructions which apply to all or most of the Pro II modules:

- [P2\\_Check\\_LED](#) (page 5)
- [P2\\_Set\\_LED](#) (page 6)
- [P2\\_Event\\_Enable](#) (page 7)
- [P2\\_Event\\_Config](#) (page 8)
- [P2\\_Event2\\_Config](#) (page 9)
- [P2\\_Event\\_Read](#) (page 11)
- [CPU\\_Digin](#) (page 12)
- [CPU\\_Digout](#) (page 13)
- [CPU\\_Dig\\_IO\\_Config](#) (page 14)
- [CPU\\_Event\\_Config](#) (page 15)
- [P2\\_Sync\\_All](#) (page 16)
- [P2\\_Sync\\_Enable](#) (page 18)
- [P2\\_Sync\\_Mode](#) (page 20)
- [P2\\_Sync\\_Stat](#) (page 22)



**P2\_Check\_LED** returns the status of the LED (on top of the front panel) of the module.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Check_LED(module)
```

## Parameters

<b>module</b>	Module address (0...15): 0: CPU module. 1...15: Set module address.	LONG
<b>ret_val</b>	0: LED off (default). 1: LED on.	LONG

## Notes

- / -

## See also

[P2\\_Set\\_LED](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E, AOut-4/16 Rev. E, AOut-8/16 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, CPU-T11, DIO-32 Rev. E, DIO-32-TiCo Rev. E, EtherCAT-SL Rev. E, FlexRay-2 Rev. E, LIN-2 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, OPT-16 Rev. E, Profi-SL Rev. E Rev. E, PWM-16(-I) Rev. E, REL-16 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, RTD-8 Rev. E, TRA-16 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Init:
If (P2_Check_LED(1)=0) Then 'If LED is off ...
    P2_Set_LED(1,1)          '... switch LED on
EndIf
```

## P2\_Check\_LED

## P2\_Set\_LED

**P2\_Set\_LED** switches the LED (on top of the front panel) on or off.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_Set_LED(module,pattern)
```

### Parameters

<b>module</b>	Module address (0...15): 0: CPU module. 1...15: Selected module address.	LONG
<b>pattern</b>	Status of the LED: 0: switch off. 1: switch on.	LONG

### Notes

- / -

### See also

[P2\\_Check\\_LED](#), [P2\\_CAN\\_Set\\_LED](#), [P2\\_LIN\\_Set\\_LED](#), [P2\\_RS\\_Set\\_LED](#)

### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E, AOut-4/16 Rev. E, AOut-8/16 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, CPU-T11, DIO-32 Rev. E, DIO-32-TiCo Rev. E, EtherCAT-SL Rev. E, FlexRay-2 Rev. E, LIN-2 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, OPT-16 Rev. E, Profi-SL Rev. E Rev. E, PWM-16(-I) Rev. E, REL-16 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, RTD-8 Rev. E, TRA-16 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
  
Init:  
    P2_Set_LED(1,1)           'Switch on LED of module 1  
  
Event:  
    Rem ...  
  
Finish:  
    P2_Set_LED(1,0)           'Switch off LED of module 1  
    Rem ...
```

**P2\_Event\_Enable** enables or disables an external event input on the specified module.

With a signal at this input a cycle of an *ADbasic* process can be controlled.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Event_Enable(module,enable)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>enable</b>	0 : disable external event signal (default). 1 : enable external event signal.	LONG

## Notes

One high-priority *ADbasic* process (that is its cyclic section **Event:**), may be called by an external event signal, e.g. to synchronize it with an external process (see *ADbasic* manual).

Most of the modules have an event input. First configure the event input using **P2\_Event\_Config**. As soon as you have enabled the event input with **P2\_Event\_Enable**, the input signal will be forwarded to the processor module. The processor module recognizes the selected type of edge (rising or falling) as event signal and the specified process responds.

For modules with several event inputs note the settings done with **P2\_Event2\_Config**. The settings of **P2\_Event\_Config** will then refer to the resulting event signal.

The event input of a processor module is always active and cannot be disabled with this instruction. The event input of the other modules is disabled after power-up.

In a system only one event input may be active, in addition to a processor module, that is you have to disable an actually active event input, before you enable the event input of another module.

## See also

[P2\\_Event\\_Config](#), [P2\\_Event2\\_Config](#), [P2\\_Event\\_Read](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E, AOut-4/16 Rev. E, AOut-8/16 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32 Rev. E, DIO-32-TiCo Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, OPT-16 Rev. E, PWM-16(-I) Rev. E, REL-16 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, TRA-16 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Init:
    REM Configure event input for minimum time of 15 ns,
    REM falling edge, 4 edges
    P2_Event_Config(1,0,2,4)
    'Enable an external event at the module 1
    P2_Event_Enable(1,1)
```

## P2\_Event\_Enable

### One event input

### Several event inputs



## P2\_Event\_Config

**P2\_Event\_Config** configures the external event input of the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Event_Config(module,min_hold,edge,prescale)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>min_hold</b>	Minimum time, which an edge must be held to be accepted: 0: 15ns (Default). 1: 50ns.	LONG
<b>edge</b>	Type of edge which is accepted: 1: rising edge (Default). 2: fallig edge. 3: rising and falling edge.	LONG
<b>prescale</b>	Number (1...255) of edges, after which an event signal is triggered (Default: 1).	LONG

### Notes

An event input must be enabled with **P2\_Event\_Enable** to have a present signal processed. First configure the event input with **P2\_Event\_Config** and enable the input then.



For modules with several event inputs note the settings done with **P2\_Event2\_Config**. The settings of **P2\_Event\_Config** will then refer to the resulting event signal. The setting **min\_hold** is valid for all event inputs.

### See also

[P2\\_Event\\_Enable](#), [P2\\_Event2\\_Config](#), [P2\\_Event\\_Read](#)

### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E, AOut-4/16 Rev. E, AOut-8/16 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32 Rev. E, DIO-32-TiCo Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, OPT-16 Rev. E, PWM-16(-I) Rev. E, REL-16 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, TRA-16 Rev. E

### Example

```
#Include ADwinPro_All.Inc

Init:
    REM Configure event input for minimum time of 15 ns,
    REM falling edge, 4 edges
    P2_Event_Config(1,0,2,4)
    'Enable an external event at the module 1
    P2_Event_Enable(1,1)
```

**P2\_Event2\_Config** configures the pre-processing of event signals on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_Event2_Config (module, mode, edge)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>mode</b>	Mode of event signal pre-processing: 0: No pre-processing (default). 1: Signal after clearance at the input <b>ENABLE</b> . 2: Signal from AB mode. 3: Signal from AB mode after clearance at the input <b>ENABLE</b> .	LONG
<b>edge</b>	For <b>mode</b> =1 or <b>mode</b> =3 only; type of edge, that is accepted at the input <b>ENABLE</b> : 1: positive edge (default). 2: negative edge. 3: positive and negative edge.	LONG

## Notes

The instruction works only for modules with more than one event input. The module processes incoming event signals to the resulting event signal, which controls the event process on the processor module.

The resulting event signal is configured with **P2\_Event2\_Config** and enabled with **P2\_Event2\_Enable**.

According to the module different modes are available:

Module name	Available modes
Pro-Aln-F-8/14-D	0, 1, 2, 3
Pro-Aln-F-8/18-D	0, 1

The module passes the signal at input **EVENT / A** as resulting event signal. The parameter **edge** is of no importance.

The input **EVENT / A** is disabled first, the resulting event signal is TTL level low. As soon as an edge of type **edge** arrives at **ENABLE**, the module enables input **EVENT / A** and passes its signal as resulting event signal.

The input **EVENT / A** is disabled again by setting mode 0.

In AB mode, the module evaluates two rectangular signals at the inputs **EVENT / A** and **ENABLE**, which are phase-shifted by 90 degrees (typical for incremental encoders): If an edge of type **edge** arrives at one of the inputs, a resulting event signal is triggered.

The maximum input frequency is 5MHz; in combination with the 4 edges per signal cycle the maximum frequency of the resulting event signal enues to 20MHz.

The time between an edge at **EVENT / A** and an edge at **ENABLE** must not be shorter than 50 ns. Impulse widths or pause durations shorter than 100 ns are not processed.

Changing the phase-shift has an effect on the maximum input frequency because of the minimum time between the edges. If the phase-shift differs from 90 degrees, the maximum input frequency of 5 MHz decreases for instance to 45 degrees at 2.5 MHz

## P2\_Event2\_Config

No pre-processing

Signal after clearance

Signal from AB mode

**Signal from AB mode  
after clearance**

The inputs EVENT / A and ENABLE are disabled first, the resulting event signal is TTL level low. As soon as an edge of type **edge** arrives at B, the module enables the inputs EVENT / A and ENABLE and evaluates the two rectangular signals to the resulting event signal (see [Signal from AB mode](#)).

The inputs EVENT / A and B are disabled again by setting mode 0.

**See also**

[P2\\_Event\\_Enable](#), [P2\\_Event\\_Config](#), [P2\\_Event\\_Read](#)

**Valid for**

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

**Example**

```
#Include ADwinPro_All.Inc
```

**Init:**

```
REM Configure event input for minimum time of 15 ns,  
REM falling edge, 4 edges  
P2_Event_Config(1,0,2,4)  
Rem set event pre-processing to clearance and  
Rem negative edge  
P2_Event2_Config(1,1,2)  
REM Enable an external event at the module 1  
P2_Event_Enable(1,1)
```

**P2\_Event\_Read** returns the current TTL level at the event inputs of the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Event_Read(module)
```

## Parameters

**module** Module address (0...15): LONG  
1...15: specified module address.

**ret\_val** Bit pattern representing the current TTL levels; LONG  
mapping of bits and event inputs see table.  
Bit = 0: TTL level low.  
Bit = 1: TTL level high.

Bits in <b>ret_val</b>	31:03	02	01	00
Input	–	B	Enable	Event / A

## Notes

The inputs A, B, and Enable are not present on any module.

## See also

[P2\\_Event\\_Enable](#), [P2\\_Event\\_Config](#), [P2\\_Event2\\_Config](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E, AOut-4/16 Rev. E, AOut-8/16 Rev. E, CAN-2 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32 Rev. E, DIO-32-TiCo Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, OPT-16 Rev. E, PWM-16(-I) Rev. E, REL-16 Rev. E, RSxxx-2 Rev. E, RSxxx-4 Rev. E, TRA-16 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Init:
    Rem Configure event input of module 1 (Aln-F-8/14)
    Rem with min. time 15 ns, neg. edge, 4 edges
    P2_Event_Config(1,0,2,4)
    Rem enable event signal of module 1
    P2_Event_Enable(1,1)

Event:
    Par_1 = P2_Event_Read(1)
```

## P2\_Event\_Read

## CPU\_Digin

Processor T11 only. **CPU\_Digin** returns, whether a falling edge arose at the input DIG I/O of the processor module since the last call of the instruction.

### Syntax

```
#Include ADwinPro_All.Inc
ret_val = CPU_Digin(channel)
```

### Parameters

<b>channel</b>	Number of the DIG I/O input on the module: 0: DIG I/O 0. 1: DIG I/O 1.	LONG
<b>ret_val</b>	Flag, if a falling edge has been detected at the DIG I/O input: 0: No falling edge detected. 1: Falling edge has been detected at least once.	LONG

### Notes

**CPU\_Digin** is active only if the selected DIG I/O channel is configured as input with **CPU\_Dig\_IO\_Config**.

Using **CPU\_Dig\_IO\_Config** you set, whether **CPU\_Digin** reacts on a rising or a falling edge. After startup the DIG I/O channels are configured as inputs and for falling edges.

**CPU\_Digin** reads the module's internal flag for falling edges; doing so, the flag will be automatically reset to the value 0.

The inputs DIG I/O work with TTL signals only.

### See also

[CPU\\_Digout](#), [CPU\\_Dig\\_IO\\_Config](#), [CPU\\_Digin](#) (T9, T10)

### Valid for

CPU-T11

### Example

```
#Include ADwinPro_All.Inc
Dim dummy As Long
```

#### Init:

```
REM Set both DIG I/O channels as input with rising edge
CPU_Dig_IO_Config(100010b)
REM Read and thus reset status signal on DIG I/O 1
dummy = CPU_Digin(1)
```

#### Event:

```
Rem ...
If (CPU_Digin(1) = 1) Then 'If falling edge has been detected ...
    End '... end this program
EndIf
Rem ...
```



**CPU\_Digout** sets a DIG I/O output of the processor module to the selected TTL level.

## Syntax

```
#Include ADwinPro_All.Inc
CPU_Digout(channel, level)
```

## Parameters

<b>channel</b>	Number (0, 1) of the DIG I/O output on the processor module.	LONG
<b>level</b>	TTL level of the output: 0: TTL level low. 1: TTL level high.	LONG

## Notes

**CPU\_Digout** is active only if the selected DIG I/O channel is configured as output with **CPU\_Dig\_IO\_Config**.

## See also

[CPU\\_Digin](#), [CPU\\_Dig\\_IO\\_Config](#)

## Valid for

CPU-T11

## Example

```
#Include ADwinPro_All.Inc
```

### Event:

```
Rem ...
CPU_Digout(1,0)           'Set DIG I/O 1 to TTL level low
Rem ...
```

## CPU\_Digout

## CPU\_Dig\_IO\_Config

**CPU\_Dig\_IO\_Config** configures all DIG I/O channels of the processor module.

### Syntax

```
#Include ADwinPro_All.Inc
CPU_Dig_IO_Config(pattern)
```

### Parameters

**pattern** Bit pattern for setting the type of channel and edge LONG at the inputs DIG I/O n:  
 Bit = 0: Channel as input or falling edge.  
 Bit = 1: Channel as output or rising edge.

Bits in <b>pattern</b>		31:06	05	04	03:02	01	00
DIG I/O 0	Type of channel	–	–	–	–	–	x
	Type of edge	–	–	–	–	x	–
DIG I/O 1	Type of channel	–	–	x	–	–	–
	Type of edge	–	x	–	–	–	–

### Notes

The type of edge may be set for inputs only.

After startup the DIG I/O channels are configured as inputs and for falling edges.

### See also

[CPU\\_Digin](#), [CPU\\_Digout](#), [CPU\\_Event\\_Config](#)

### Valid for

CPU-T11

### Example

```
#Include ADwinPro_All.Inc
Dim dummy As Long
```

#### Init:

```
REM Set REM Set both DIG I/O channels as input with rising edge
CPU_Dig_IO_Config(100010b)
REM Read and thus reset status signal on DIG I/O 1
dummy = CPU_Digin(1)
Rem ...
```

**CPU\_Event\_Config** configures the Event In channel of the processor module.

## Syntax

```
#Include ADwinPro_All.Inc

CPU_Event_Config(min_hold, edge, prescale)
```

## Parameters

<b>min_hold</b>	Minimum time, which an edge must be held to be accepted: 0: 15ns (Default). 1: 50ns.	LONG
<b>edge</b>	Type of edge which is accepted: 1: rising edge (Default). 2: fallig edge. 3: ising and falling edge.	LONG
<b>prescale</b>	Number (1...15) of edges, after which an event signal is triggered (Default: 1).	LONG

## Notes

The input Event In works with TTL signals only.

If input signals contain glitches - as far as can't be avoided - you may do the following:

- Set parameter **min\_hold** to 1, to filter glitches.
- Redirect the input signal via an opto couple first.
- Connect the input signal t the module Pro-OPT-16 and enable the module's external event input with **EventEnable**.

## See also

[P2\\_Event\\_Enable](#), [P2\\_Event\\_Read](#), [EventEnable](#)

## Valid for

CPU-T11

## Example

```
#Include ADwinPro_All.Inc
```

### Init:

```
REM Configure input EVENT IN for mimimum time of 15 ns,
REM falling edge, 4 edges
CPU_Event_Config(0,2,4)
```

### Event:

```
REM Externally controlled process starts each time, when
REM 4 falling edges have reached the input EVENT IN.
Rem ...
```

## CPU\_Event\_Config

## P2\_Sync\_All

**P2\_Sync\_All** starts a specified action synchronically on the selected modules.

### Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_Sync_All(pattern)
```

### Parameters

**pattern** Bit pattern selecting the addresses of the modules LONG which start an action:  
 Bit = 0: Ignore module.  
 Bit = 1: Start module action synchronously.

Bits in <b>pattern</b>	31:16	14	13	...	01	00
Module address	–	15	14	...	2	1

### Notes

The action starting on the selected modules depends on the module types. The configurations apply you have made before for the multiplexer, output value etc.

Module type	Action
Analog input	Start A/D conversion on all enabled ADCs, see <a href="#">P2_START_CONV</a> / <a href="#">P2_START_CONV_F</a> . The conversion can be a single conversion or part of a burst sequence.
Analog output	Start D/A conversion on all enabled DACs with the value of the DAC register, see <a href="#">P2_START_DAC</a> .
Digital input	Transfer current status of the inputs into the input temporary register (read out the values there with <a href="#">P2_DIG_READ_LATCH</a> ).
Digital output	Read the value from the output temporary register and transfer it to the digital output (see <a href="#">P2_DIG_WRITE_LATCH</a> ).
Counter	Current counter values are copied into counter latches; read from there e.g. with <a href="#">P2_CNT_READ_LATCH</a> or <a href="#">P2_CNT_GET_PW</a> .

As default all inputs / outputs of the selected modules participate in the action. Using **P2\_Sync\_Enable** you may disable or enable one or more inputs / outputs of a module for synchronization.

The following instructions do a synchronous action, too:

- **P2\_Sync\_Mode**: An external event signal triggers a conversion simultaneously on all channels. The conversion can be part of a single measurement (**P2\_ADCF\_Mode**) or of a burst-measurement sequence (**P2\_Burst\_Init**).
- **P2\_Burst\_Start**: By software, burst sequences are started on several modules.

### See also

[P2\\_Sync\\_Enable](#), [P2\\_Sync\\_Mode](#), [P2\\_Sync\\_Stat](#)

[P2\\_ADCF\\_Mode](#), [P2\\_Burst\\_Start](#), [P2\\_Start\\_Conv](#), [P2\\_Start\\_Conv\\_F](#), [P2\\_Start\\_DAC](#)

P2\_Write\_DAC, P2\_Write\_DAC8, P2\_Write\_DAC8\_Packed, P2\_Write\_DAC32

P2\_Cnt\_Get\_PW, P2\_Cnt\_Read\_Latch, P2\_Cnt\_Sync\_Latch

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E, AOut-4/16 Rev. E, AOut-8/16 Rev. E, CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, DIO-32 Rev. E, DIO-32-TiCo Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, OPT-16 Rev. E, PWM-16(-I) Rev. E, REL-16 Rev. E, TRA-16 Rev. E

## Example

```
#Include ADwinPro_All.Inc
```

```
Dim i As Long
```

```
Dim Data_1[1000], Data_2[1000], Data_3[1000] As Long
```

```
Dim Data_5[1000] As Long
```

### Init:

```
REM Enable channel synchronization on the modules 1,2,4,5
```

```
P2_Sync_Enable(1,11b)
```

```
P2_Sync_Enable(2,11b)
```

```
P2_Sync_Enable(4,11b)
```

```
P2_Sync_Enable(5,100b)
```

```
P2_Write_DAC(5,1,0) 'initialize output
```

```
i=1 'initialize index
```

### Event:

```
REM Start conversion of modules 1, 2, 4 and 5 synchronously
```

```
P2_Sync_All(11011b)
```

```
P2_Wait_EOC(1) 'Wait for end of conversion
```

```
Rem Read A/D converter 1 of modules 1,2,4
```

```
Data_1[i]=P2_Read_ADCF(1,1)
```

```
Data_2[i]=P2_Read_ADCF(2,1)
```

```
Data_3[i]=P2_Read_ADCF(4,1)
```

```
REM write value into output register of D/A module 5
```

```
P2_Write_DAC(5,1,Data_5[i])
```

```
If (i=1000) Then End 'End process after 1000 repetitions
```

```
Inc(i) 'Increment index
```

### P2\_Sync\_Enable

**P2\_Sync\_Enable** enables or disables the synchronizing option for selected inputs, outputs or function groups on the specified module.

#### Syntax

```
#Include ADwinPro_All.Inc

P2_Sync_Enable(module, channel)
```

#### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>channel</b>	Bit pattern for selection of inputs, outputs or function groups that are enabled or disabled: 0 : disable. 1: enable.	LONG

Bits in <b>channel</b>	31:08	07	06	05	04	03	02	01	00
Channel no. on analog input module	–	8	7	6	5	4	3	2	1
Aln-F-x/x Rev. E									
Channel no. on analog output module	–	8	7	6	5	4	3	2	1
Bits in <b>channel</b>	31:04	03	02	01	00				
Function group in Multi-I/O modules	–	counters	analog inputs	analog outputs	digital outputs				

#### Notes

The default setting after start-up for all modules is the value **0FFFFh**, that is all inputs, outputs or function groups are enabled.

The instruction sets all channels or function groups of a module at the same time. If you want to disable or enable a single channel, you have to indicate the settings of the other channels as well.

The synchronizing signal is triggered by **P2\_Sync\_All**.

#### See also

[P2\\_Sync\\_All](#), [P2\\_Sync\\_Mode](#), [P2\\_Sync\\_Stat](#)

#### Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E, AOut-4/16 Rev. E, AOut-8/16 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, PWM-16(-I) Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim i As Long
Dim Data_1[1000], Data_2[1000], Data_3[1000] As Long
Dim Data_5[1000] As Long

Init:
    REM Enable channel synchronization on the modules 1,2,4,5
    P2_Sync_Enable(1,11b)
    P2_Sync_Enable(2,11b)
    P2_Sync_Enable(4,11b)
    P2_Sync_Enable(5,100b)
    P2_Write_DAC(5,1,0)      'initialize output
    i=1                      'initialize index

Event:
    REM Start conversion of modules 1, 2, 4 and 5 synchronously
    P2_Sync_All(11011b)
    P2_Wait_EOC(1)           'Wait for end of conversion
    Rem Read A/D converter 1 of modules 1,2,4
    Data_1[i]=P2_Read_ADCF(1,1)
    Data_2[i]=P2_Read_ADCF(2,1)
    Data_3[i]=P2_Read_ADCF(4,1)

    REM write value into output register of D/A module 5
    P2_Write_DAC(5,1,Data_5[i])
    If (i=1000) Then End     'End process after 1000 repetitions
    Inc(i)                   'Increment index
```

## P2\_Sync\_Mode

**P2\_Sync\_Mode** enables or disables the synchronization (of conversions) with other modules as master or as slave.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Sync_Mode(module,mode)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>mode</b>	Synchronization mode of the module (0...2): 0: no synchronization (default setting). 1: Synchronization as master module. 2: Synchronization as slave module.	LONG

### Notes

The synchronization allows to have a signal at the event input of the master module simultaneously start conversions on all slave modules. The master module will forward the event signal to the slave modules.

The settings for pre-processing the event signal (**P2\_Event\_Config**, **P2\_Event2\_Config**) may be different for each module.

Only one master module is allowed.

As soon as synchronized slave modules (mode 2) receive the forwarded event signal, they start a conversion (like **P2\_Start\_ConvF** does) simultaneously on all channels. The conversion can be part of a single measurement or of a burst-measurement sequence.

If you synchronize burst-measurement sequences on several modules, you should initialize the modules to the same number of measurements with **P2\_Burst\_Init**. This refers especially to the master / slave synchronization: The number of measurements on the master module must be equal or greater than the number on the slave modules. If not, on the latter modules the burst-measurement sequence will not be ended with the last signal from the master module.

### See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_Read](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#), [P2\\_Event\\_Enable](#), [P2\\_Event\\_Config](#), [P2\\_Event2\\_Config](#), [P2\\_Start\\_ConvF](#), [P2\\_Sync\\_All](#), [P2\\_Sync\\_Enable](#), [P2\\_Sync\\_Stat](#)

### Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E



### Example

```
#Include ADwinPRO_ALL.Inc
#Define count 10000
#Define module 1

Dim i As Long
Dim Data_1[count], Data_2[count], Data_3[count] As Long
Dim Data_4[count], Data_5[count], Data_6[count] As Long
Dim Data_7[count], Data_8[count], Data_9[count] As Long
Dim Data_10[count], Data_11[count], Data_12[count] As Long

Init:
P2_Sync_Mode(module,1) 'master module
P2_Sync_Mode(module+1,2) 'slave module 1
P2_Sync_Mode(module+2,2) 'slave module 2
Rem initialize and start burst measurement for 4 channels
P2_Burst_Init(module,15,0,count,1,100b)
P2_Burst_Init(module+1,15,0,count,1,100b)
P2_Burst_Init(module+2,15,0,count,1,100b)
P2_Burst_Start(111b) 'start burst measurement on module 1-3
Processdelay=800 'get trigger point with 50 kHz

Event:
Par_1=P2_Burst_Status(module)'number of remaining measurements
If (Par_1=0) Then End 'burst sequence finished, go to FINISH

Finish:
Rem copy the last converted data of all 4 channels
P2_Burst_Read_Unpacked4(module,count,0,
    Data_1,Data_2,Data_3,Data_4,1,3)
P2_Burst_Read_Unpacked4(module+1,count,0,
    Data_5,Data_6,Data_7,Data_8,1,3)
P2_Burst_Read_Unpacked4(module+2,count,0,
    Data_10,Data_10,Data_11,Data_12,1,3)
```

**P2\_Sync\_Stat**

**P2\_Sync\_Stat** returns the settings of the synchronizing option of the specified module.

**Syntax**

```
#Include ADwinPro_All.Inc

ret_val = P2_Sync_Stat(module)
```

**Parameters**

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Setting of the synchronizing option at the inputs / outputs: 0 : disabled. 1 : enabled.	LONG

Bits in <b>channel</b>	31:08	07	06	05	04	03	02	01	00
Channel no. in analog inpt modules Aln-F-x/x Rev. E	–	8	7	6	5	4	3	2	1
Channel no. in analog outpt modules AOut-x/16 Rev. E	–	8	7	6	5	4	3	2	1
Bits in <b>channel</b>	31:04	03	02	01	00				
Function group in Multi-I/O modules	–	counters	analog inputs	analog outputs	digital outputs				

**Notes**

You set the synchronizing option of the channels or function groups with **P2\_Sync\_Enable**.

**See also**

[P2\\_Sync\\_All](#), [P2\\_Sync\\_Enable](#), [P2\\_Sync\\_Mode](#)

**Valid for**

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E, AOut-4/16 Rev. E, AOut-8/16 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, PWM-16(-I) Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim i As Long
Dim Data_1[1000], Data_2[1000] As Long

Init:
  REM Is channel 1 on module 1 still disabled?
  If (P2_Sync_Stat(1) And 1 = 0) Then
    REM Enable channel 1 on D/A modules 1+2
    REM disable all other channels
    P2_Sync_Enable(1,1)
    P2_Sync_Enable(2,1)
  EndIf
  i=1                                'Initialize index

Event:
  REM write values into output registers
  P2_Write_DAC(1,1,Data_1[i])
  P2_Write_DAC(2,1,Data_2[i])
  REM Start output on modules 1+2 synchronously
  P2_Sync_All(11b)
  If (i=1000) Then End              'End process after 1000 repetitions
  Inc(i)                           'Increment index
```

### 3.2 Pro II: Multi-I/O

This section describes instructions which apply to Pro II multi I/O modules:

- [P2\\_MIO\\_Dig\\_Latch](#) (page 25)
- [P2\\_MIO\\_Dig\\_Read\\_Latch](#) (page 26)
- [P2\\_MIO\\_Dig\\_Write\\_Latch](#) (page 27)
- [P2\\_MIO\\_Digin\\_Long](#) (page 28)
- [P2\\_MIO\\_Digout](#) (page 29)
- [P2\\_MIO\\_Digout\\_Long](#) (page 30)
- [P2\\_MIO\\_DigProg](#) (page 31)
- [P2\\_MIO\\_Get\\_Digout\\_Long](#) (page 32)

In the Instruction List sorted by Module Types (annex A.2) you will find overviews of the instructions corresponding to the *ADwin-Pro* modules.

It is presumed that application examples use the module address 1 for D/A modules.



**P2\_MIO\_Dig\_Latch** transfers digital information from the inputs to the input latches and/or from the output latches to the outputs on the specified module.

## Syntax

```
#Include ADwinPro_All.inc

P2_MIO_Dig_Latch(module)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
---------------	------------------------------------	------

## Notes

It is recommended to first program the channels as inputs or outputs using **P2\_MIO\_DigProg**, except for TRA and OPT channels.

With digital inputs the instructions reads the input signals into the input latches. With digital outputs the instruction passes the values of the output latches to the outputs.

If the module is released for synchronization by **P2\_Sync\_Enable**, **P2\_Sync\_All** has the same functions as **P2\_MIO\_Dig\_Latch**.

## See also

[P2\\_MIO\\_Dig\\_Read\\_Latch](#), [P2\\_MIO\\_Dig\\_Write\\_Latch](#), [P2\\_MIO\\_DigProg](#), [P2\\_MIO\\_Digin\\_Long](#), [P2\\_MIO\\_Digout](#), [P2\\_MIO\\_Digout\\_Long](#), [P2\\_MIO\\_Get\\_Digout\\_Long](#), [P2\\_Sync\\_All](#)

## Valid for

MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.inc
```

### Init:

```
Rem set channels 0...3 as outputs, 4...7 as inputs
P2_MIO_DigProg(1,01b)
P2_MIO_Dig_Write_Latch(1,0)'set output bits to 0
```

### Event:

```
Rem latch inputs and output the content of output latches
P2_MIO_Dig_Latch(1)
Par_1 = P2_MIO_Dig_Read_Latch(1) 'read input bits and ...
P2_MIO_Dig_Write_Latch(1,Par_1)'output with next event
```

## P2\_MIO\_Dig\_Latch

## P2\_MIO\_Dig\_Read\_Latch

**P2\_MIO\_Dig\_Read\_Latch** returns the bits from the latch register for the digital inputs of the specified module.

### Syntax

```
#Include ADwinPro_All.inc

ret_val = P2_MIO_Dig_Read_Latch(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Bit pattern of the latch register. Each bit corresponds to a digital input (see table).	LONG

Bit no.	31...20	19 ... 16	15...8	7 6 ... 1 0
Input	–	OPT4 ... OPT1	–	7 6 ... 1 0

### Notes

It is recommended to first program the specified channels as inputs using **P2\_MIO\_Digprog**, except for OPT inputs.

The current status of the digital inputs can be transferred to the latch register with the following instructions:

- **P2\_MIO\_Dig\_Latch**
- **P2\_Sync\_All** (when activated)

### See also

[P2\\_MIO\\_Dig\\_Latch](#), [P2\\_MIO\\_Dig\\_Write\\_Latch](#), [P2\\_MIO\\_DigProg](#),  
[P2\\_MIO\\_Digin\\_Long](#), [P2\\_Sync\\_All1Valid](#) for  
[MIO-4 Rev. E](#)

### Example

```
#Include ADwinPro_All.inc
Dim value As Long
```

#### Init:

```
Rem set channels 07:00 of modules 1+2 as inputs
P2_MIO_Digprog(1,00b)
P2_MIO_Digprog(2,00b)
```

#### Event:

```
Rem copy digital output levels of both modules synchronously
Rem into the latches
P2_Sync_All(11b)
Par_1 = P2_MIO_Dig_Read_Latch(1)'read latch of module 1
Par_2 = P2_MIO_Dig_Read_Latch(2)'read latch of module 2
```

**P2\_MIO\_Dig\_Write\_Latch** writes a 32 bit value into the latch register for the digital outputs on the specified module.

## Syntax

```
#Include ADwinPro_All.inc

P2_MIO_Dig_Write_Latch(module,pattern)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern. Each bit corresponds to a digital output (see table).	LONG

Bit no.	31...28	27 ... 24	23...8	7 6 ... 1 0
Output	–	TRA4 ... TRA1	–	7 6 ... 1 0

## Notes

The specified channels must first be programmed as outputs using **P2\_MIO\_Digprog**, except for TRA outputs.

You may set the value of the latch register for digital outputs with **P2\_MIO\_Digout**.

## See also

[P2\\_MIO\\_Dig\\_Latch](#), [P2\\_MIO\\_Digout](#), [P2\\_MIO\\_DigProg](#), [P2\\_MIO\\_Get\\_Digout\\_Long](#)

## Valid for

MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.inc

Init:
    P2_MIO_Digprog(1,11b)    'module channels 07:00 as outputs

Event:
    Rem output information of output latch to the pins
    P2_MIO_Dig_Latch(1)
    Rem write a long word to the output latch
    P2_MIO_Dig_Write_Latch(1,Par_1)
```

## P2\_MIO\_Dig\_Write\_Latch

## P2\_MIO\_Digin\_Long

**P2\_MIO\_Digin\_Long** returns the status of the inputs (bits 7...0) of the specified module as bit pattern

### Syntax

```
#Include ADwinPro_All.inc

ret_val = P2_MIO_Digin_Long(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Bit pattern. Each bit (7...0) corresponds to the input status of a digital input (see table). Bit = 0: Input has low level. Bit = 1: Input has high level.	LONG

Bit no.	31...20	19 ... 16	15...8	7 6 ... 1 0
Input	–	OPT4 ... OPT1	–	7 6 ... 1 0

### Notes

It is recommended to first program the specified channels as inputs using **P2\_MIO\_Digprog**, except for OPT inputs.

### See also

[P2\\_MIO\\_Dig\\_Latch](#), [P2\\_MIO\\_DigProg](#), [P2\\_MIO\\_Digout\\_Long](#)

### Valid for

MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.inc

Init:
    P2_MIO_Digprog(1,00b)    'channels 07:00 as inputs

Event:
    Par_1 = P2_MIO_Digin_Long(1)'read all inputs
```



**P2\_MIO\_Digout** sets a single output of the specified module to the level "high" or "low". All other outputs remain unchanged.

## Syntax

```
#Include ADwinPro_All.inc

P2_MIO_Digout (module, output, value)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>output</b>	Number of the output to be set. 0...7: DIO channels. 24...27: TRA outputs.	LONG
<b>value</b>	New status of the selected output: 0: Low level. 1: High level.	LONG

## Notes

The specified channels must be first programmed as outputs using **P2\_MIO\_Digprog**; except for TRA outputs.

With **P2\_MIO\_Digout**, you can set or clear any output without changing the status of the remaining outputs.

## See also

[P2\\_MIO\\_Digout\\_Long](#) [P2\\_MIO\\_DigProg](#)

## Valid for

MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.inc
```

### Init:

```
Rem channels 0...3 as inputs, 4...7 as outputs
P2_MIO_Digprog(1,10b)
```

### Event:

```
Rem read input bits and check if channel 3 is set
If (P2_MIO_Digin_Long(1) And 1000b = 1) Then
    P2_MIO_Digout(1,5,0)    'channel 3 is set: clear bit 5
Else
    P2_MIO_Digout(1,5,1)    'channel 3 is set: set bit 5
EndIf
```

## P2\_MIO\_Digout

## P2\_MIO\_Digout\_Long

**P2\_MIO\_Digout\_Long** sets or clears all outputs on the specified module.

### Syntax

```
#Include ADwinPro_All.inc

P2_MIO_Digout_Long(module,pattern)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern that sets the digital outputs: Bit = 0: Set output to level "low". Bit = 1: Set output to level "high".	LONG

Bit no.	31...28	27 ... 24	23...8	7 6 ... 1 0
Output	–	TRA4 ... TRA1	–	7 6 ... 1 0

### Notes

The specified channels must be first programmed as outputs using **P2\_MIO\_Digprog**, except for TRA outputs.

### See also

[P2\\_MIO\\_Digout](#), [P2\\_MIO\\_DigProg](#)

### Valid for

MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.inc

Init:
    P2_MIO_Digprog(1,01111b) 'channels 07:00 as outputs

Event:
    P2_MIO_Digout_Long(1,128)'Output 128 as binary value on the
                           'channels
```

**P2\_MIO\_DigProg** programs the digital channels 0...7 of the specified module as inputs or outputs in groups of 4.

## Syntax

```
#Include ADwinPro_All.inc

P2_MIO_Digprog(module, pattern)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern that sets the channels as inputs or outputs: Bit = 0: Set channel as input. Bit = 1: Set channel as output.	LONG

Bit no.	31...2	1	0
channel no.	–	07:04	03:00

## Notes

After power-up of the system all channels are configured as inputs.

Channels can only be set as inputs or outputs in groups of 4, 2 relevant bits only, the other bits are ignored.

## See also

[P2\\_MIO\\_Dig\\_Latch](#), [P2\\_MIO\\_Dig\\_Read\\_Latch](#), [P2\\_MIO\\_Dig\\_Write\\_Latch](#)

[P2\\_MIO\\_Digin\\_Long](#), [P2\\_MIO\\_Digout](#), [P2\\_MIO\\_Digout\\_Long](#), [P2\\_MIO\\_Get\\_Digout\\_Long](#)

## Valid for

MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.inc
```

### Init:

```
Rem Configure channels 0...3 of module no. 1 as inputs
Rem and channels 4...7 as outputs
P2_MIO_Digprog(1, 10b)
```

## P2\_MIO\_DigProg

## P2\_MIO\_Get\_Digout\_Long

**P2\_MIO\_Get\_Digout\_Long** returns the contents of the output latch (register for digital outputs) on the specified module.

### Syntax

```
#Include ADwinPro_All.inc

ret_val = P2_MIO_Get_Digout_Long(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Contents of the output latch. Each bit corresponds to the input status of a digital input (see table): Bit = 0: Output has low level. Bit = 1: Output has high level.	LONG

Bit no.	31...28	27 ... 24	23...8	7 6 ... 1 0
Output	–	TRA4 ... TRA1	–	7 6 ... 1 0

### Notes

Returning the current status of the outputs instead of the output latch is technically impossible.

### See also

[P2\\_MIO\\_Dig\\_Latch](#), [P2\\_MIO\\_Dig\\_Read\\_Latch](#), [P2\\_MIO\\_Dig\\_Write\\_Latch](#)

[P2\\_MIO\\_DigProg](#), [P2\\_MIO\\_Digin\\_Long](#), [P2\\_MIO\\_Digout](#), [P2\\_MIO\\_Digout\\_Long](#)

### Valid for

MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.inc
```

#### Event:

```
Rem read return bits 31:00 from the latch
Par_1 = P2_MIO_Get_Digout_Long(1)
```

### 3.3 Pro II: Analog Input Modules

This section describes instructions which apply to Pro II analog input modules:

- [P2\\_ADC](#) (page 35)
- [P2\\_ADC24](#) (page 36)
- [P2\\_ADC\\_Read\\_Limit](#) (page 37)
- [P2\\_ADC\\_Set\\_Limit](#) (page 39)
- [P2\\_Read\\_ADC](#) (page 40)
- [P2\\_Read\\_ADC24](#) (page 41)
- [P2\\_Read\\_ADC\\_SConv](#) (page 42)
- [P2\\_Read\\_ADC\\_SConv24](#) (page 43)
- [P2\\_SE\\_Diff](#) (page 44)
- [P2\\_Seq\\_Init](#) (page 45)
- [P2\\_Seq\\_Read](#) (page 48)
- [P2\\_Seq\\_Read24](#) (page 49)
- [P2\\_Seq\\_Read\\_Packed](#) (page 51)
- [P2\\_Seq\\_Start](#) (page 52)
- [P2\\_Seq\\_Wait](#) (page 53)
- [P2\\_Set\\_Mux](#) (page 54)
- [P2\\_Start\\_Conv](#) (page 55)
- [P2\\_Wait\\_EOC](#) (page 56)
- [P2\\_Wait\\_Mux](#) (page 57)
- [P2\\_Burst\\_CRead\\_Unpacked1](#) (page 58)
- [P2\\_Burst\\_CRead\\_Unpacked2](#) (page 60)
- [P2\\_Burst\\_CRead\\_Unpacked4](#) (page 62)
- [P2\\_Burst\\_CRead\\_Unpacked8](#) (page 64)
- [P2\\_Burst\\_Init](#) (page 66)
- [P2\\_Burst\\_Read\\_Index](#) (page 69)
- [P2\\_Burst\\_Read](#) (page 71)
- [P2\\_Burst\\_Read\\_Unpacked1](#) (page 73)
- [P2\\_Burst\\_Read\\_Unpacked2](#) (page 75)
- [P2\\_Burst\\_Read\\_Unpacked4](#) (page 77)
- [P2\\_Burst\\_Read\\_Unpacked8](#) (page 79)
- [P2\\_Burst\\_Reset](#) (page 81)
- [P2\\_Burst\\_Start](#) (page 83)
- [P2\\_Burst\\_Status](#) (page 84)
- [P2\\_Burst\\_Stop](#) (page 86)
- [P2\\_Set\\_Average\\_Filter](#) (page 88)
- [P2\\_ADCF](#) (page 89)
- [P2\\_ADCF24](#) (page 90)
- [P2\\_ADCF\\_Mode](#) (page 91)
- [P2\\_ADCF\\_Read\\_Limit](#) (page 94)
- [P2\\_ADCF\\_Set\\_Limit](#) (page 95)

- [P2\\_ADCF\\_Reset\\_Min\\_Max](#) (page 96)
- [P2\\_ADCF\\_Read\\_Min\\_Max4](#) (page 97)
- [P2\\_ADCF\\_Read\\_Min\\_Max8](#) (page 99)
- [P2\\_Read\\_ADCF](#) (page 101)
- [P2\\_Read\\_ADCF24](#) (page 102)
- [P2\\_Read\\_ADCF4](#) (page 103)
- [P2\\_Read\\_ADCF4\\_24B](#) (page 104)
- [P2\\_Read\\_ADCF8](#) (page 105)
- [P2\\_Read\\_ADCF8\\_24B](#) (page 106)
- [P2\\_Read\\_ADCF4\\_Packed](#) (page 107)
- [P2\\_Read\\_ADCF8\\_Packed](#) (page 108)
- [P2\\_Read\\_ADCF32](#) (page 109)
- [P2\\_Read\\_ADCF\\_SConv](#) (page 110)
- [P2\\_Read\\_ADCF\\_SConv24](#) (page 111)
- [P2\\_Read\\_ADCF\\_SConv32](#) (page 112)
- [P2\\_Set\\_Gain](#) (page 113)
- [P2\\_Start\\_ConvF](#) (page 114)
- [P2\\_Wait\\_EOCF](#) (page 115)

In the [Instruction List sorted by Module Types](#) (annex A.2) you will find which of the functions corresponds to the *ADwin-Pro II* modules.

It is presumed that application examples use the module address 1 for A/D modules.



**P2\_ADC** runs a complete conversion on an ADC of the specified module. The return value has a resolution of 16 bit.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_ADC(module, input_no)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>input_no</b>	Number of the analog input (1...8 or 1...32).	LONG
<b>ret_val</b>	Result of the conversion (0...65535).	LONG

## Notes

P2\_ADC is characterized by a sequence of several instructions:

<b>P2_Set_Mux</b>	→	...	→	<b>P2_Start_Conv</b>	→	<b>P2_Wait_EOC</b>	→	<b>P2_Read_ADC</b>
set multiplexer to an input channel		wait for multiplexer settling		Start A/D conversion		Wait for end of conversion		Read out the converted value

If the multiplexer is set to the same channel as the previous conversion, the settling time is skipped automatically.

If you want to set the gain, use **P2\_Set\_Mux**.

## See also

[P2\\_ADC24](#), [P2\\_ADC\\_Read\\_Limit](#), [P2\\_ADC\\_Set\\_Limit](#), [P2\\_Read\\_ADC](#), [P2\\_Start\\_Conv](#), [P2\\_Wait\\_EOC](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim value As Long
```

### Event:

```
value = P2_ADC(1, 4) 'maesure 16Bit value at analog input 4
```

## P2\_ADC

## P2\_ADC24

**P2\_ADC24** runs a complete conversion on an ADC of the specified module. The return value is formatted to 24 bit.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_ADC24(module, input_no)
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>input_no</b>	Number of the analog input (1...8 or 1...32).	LONG
<b>ret_val</b>	Result of the conversion (0...16777215 = $2^{24}-1$ ).	LONG

### Notes

**P2\_ADC24** is characterized by a sequence of several instructions:

P2_Set_Mux	→	...	→	P2_Start_Conv	→	P2_Wait_EOC	→	P2_Read_ADC24
set multiplexer to an input channel		wait for multiplexer settling		Start A/D conversion		Wait for end of conversion		Read out the converted value

If the multiplexer is set to the same channel as the previous conversion, the settling time is skipped automatically.

If you want to set the gain, use **P2\_Set\_Mux**.

If a measurement value has a resolution less than 24 bit, the "missing" bits in the return value „fehlenden“ are filled with zeros.

As an example, the measurement value of an 18 bit ADC is located in the bits 6...23 of the return value; the measurement value is shifted to the left by 6 bits and the bits 0...5 are zeros.

Bit no.	31...24	23...6	05...00
Content	0	18 bit meas. value	0

### See also

[P2\\_ADC](#), [P2\\_ADC\\_Read\\_Limit](#), [P2\\_ADC\\_Set\\_Limit](#), [P2\\_Read\\_ADC24](#), [P2\\_Start\\_Conv](#), [P2\\_Wait\\_EOC](#)

### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim value As Long
```

#### Event:

```
REM measure 24Bit value at analog input 4
value = P2_ADC24(1, 4)
```



**P2\_ADC\_Read\_Limit** returns the flags of limit-overflow and -underrun from 16 ADCs of the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_ADC_Read_Limit(module, ch_group)
```

## Parameters

**module** Selected module address (1...15). LONG

**ch\_group** Group of 16 channels each:  
1: Channels 1...16  
2: Channels 17...32

**ret\_val** Bit pattern representing the limit-overflow and -underrun flags: LONG

overflow of upper limit								
ch_group	Bit No.	31	30	29	...	18	17	16
1	Channel no.	16	15	14	...	3	2	1
2	Channel no.	32	31	30	...	19	18	17
underrun of lower limit								
ch_group	Bit No.	15	14	13	...	2	1	0
1	Channel no.	16	15	14	...	3	2	1
2	Channel no.	32	31	30	...	19	18	17

## Notes

The limits are set with **P2\_ADC\_Set\_Limit**.

Reading the flags resets all flags of a channel group to zero.

We recommend to read the flags in the **Init:** section once, as to ensure all flags be reset. This is even more important with an externally controlled process.

## See also

[P2\\_ADC](#), [P2\\_ADC24](#), [P2\\_ADC\\_Set\\_Limit](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## P2\_ADC\_Read\_Limit

### Example

```
#Include ADwinPro_All.Inc
Dim flags As Long

Init:
    P2_SE_Diff(1,1)           'Differential inputs
    P2_ADC_Set_Limit(1, 2, 42768,256) 'Set limits for channel 2
    P2_Seq_Init(1,3,0,10b,0) 'continuous max mode, Kanal 2
    P2_Seq_Start(1)           'Start sequence control
    P2_Seq_Wait(1)
    flags = P2_ADC_Read_Limit(1,1) 'Reset flags by reading

Event:
    flags = P2_ADC_Read_Limit(1,1) 'read flags of channels 1...16
    If ((flags And 10b) = 10b) Then
        REM limit-underrun on channel 2
        Inc Par_1
    EndIf
    If ((flags And 20000h) = 20000h) Then
        REM limit-overflow on channel 2
        Inc Par_2
    EndIf
```

**P2\_ADC\_Set\_Limit** sets the upper and lower limit for one F-ADC of the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_ADC_Set_Limit(module, input_no, high, low)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>input_no</b>	Number of the analog input (1...8 or 1...32).	LONG
<b>high</b>	Upper limit (0...65535) of the channel. Default: 65535.	LONG
<b>low</b>	Lower limit (0...65535) of the channel. Default: 0.	LONG

## Notes

If a converted value exceeds the upper limit, the channel's flag is set. **P2\_ADC\_Read\_Limit** reads and thus resets the flags.

The same way a channel's flag is set for a converted value falling below the lower limit.

A limit-overflow or -underrun does not trigger an event signal.

## See also

[P2\\_ADC](#), [P2\\_ADC24](#), [P2\\_ADC\\_Read\\_Limit](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim flags As Long

Init:
P2_SE_Diff(1,1)           'Differential inputs
P2_ADC_Set_Limit(1, 2, 42768,256) 'Set limits for channel 2
P2_Seq_Init(1,3,0,10b,0) 'continuous max mode, Kanal 2
P2_Seq_Start(1)           'Start sequence control
P2_Seq_Wait(1)
flags = P2_ADC_Read_Limit(1,1) 'Reset flags by reading

Event:
flags = P2_ADC_Read_Limit(1,1) 'read flags of channels 1...16
If ((flags And 10b) = 10b) Then
    REM limit-underrun on channel 2
    Inc Par_1
EndIf
If ((flags And 20000h) = 20000h) Then
    REM limit-overflow on channel 2
    Inc Par_2
EndIf
```

## P2\_ADC\_Set\_Limit

## P2\_Read\_ADC

**P2\_Read\_ADC** reads out the conversion result from an ADC of the specified module. The return value has a resolution of 16 bit.

### Syntax

```
#Include ADwinPro_All.Inc  
  
ret_val = P2_Read_ADC(module)
```

### Parameters

module	Selected module address (1...15).	LONG
ret_val	Value from the ADC (0...65535).	LONG

### Notes

- / -

### See also

[P2\\_ADC](#), [P2\\_ADC24](#), [P2\\_Start\\_Conv](#), [P2\\_Wait\\_EOC](#), [P2\\_ADC\\_Read\\_Limit](#), [P2\\_ADC\\_Set\\_Limit](#), [P2\\_Read\\_ADC\\_SConv](#)

### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
Dim value1 As Long 'Declaration
```

#### Event:

```
P2_Set_Mux(1,0100000010b)'set MUX to input 3, gain 2  
Rem wait for multiplexer settling, here 4 µs  
P2_Sleep(400)  
P2_Start_Conv(1) 'Start AD conversion  
P2_Wait_EOC(1) 'Wait for end of conversion  
value1 = P2_Read_ADC(1) 'Read value from ADC
```

**P2\_Read\_ADC24** returns the conversion result from an ADC of the specified module. The return value has a resolution of 24 bit.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Read_ADC24(module)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>ret_val</b>	Value from the ADC ( $0 \dots 2^{24}-1$ ).	LONG

## Notes

If a measurement value has a resolution less than 24 bit, the "missing" bits in the return value „fehlenden“ are filled with zeros.

As an example, the measurement value of an 18 bit ADC is located in the bits 6...23 of the return value; the measurement value is shifted to the left by 6 bits and the bits 0...5 are zeros.

Bit no.	31...24	23...6	05...00
Content	0	18 bit meas. value	0

## See also

[P2\\_ADC24](#), [P2\\_Start\\_Conv](#), [P2\\_Wait\\_EOC](#), [P2\\_ADC\\_Read\\_Limit](#), [P2\\_ADC\\_Set\\_Limit](#), [P2\\_Read\\_ADC\\_SConv24](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim value1 As Long 'Declaration

Event:
P2_Set_Mux(1,0100000010b)'set MUX to input 3, gain 2
Rem wait for multiplexer settling, here 4 µs
P2_Sleep(400)
P2_Start_Conv(1) 'Start AD conversion
P2_Wait_EOC(1) 'Wait for end of conversion
value1 = P2_Read_ADC24(1)'Read 24 bit value from the ADC
```

## P2\_Read\_ADC24

## P2\_Read\_ADC\_SConv

**P2\_Read\_ADC\_SConv** reads out the conversion result from an ADC of the specified module and immediately starts a new conversion.

The return value has a resolution of 16 bit.

### Syntax

```
#Include ADwinPro_All.Inc  
  
ret_val = P2_Read_ADC_SConv(module)
```

### Parameters

module	Selected module address (1...15).	LONG
ret_val	Value from the ADC (0...65535).	LONG

### See also

[P2\\_ADC](#), [P2\\_ADC24](#), [P2\\_Read\\_ADC](#), [P2\\_Read\\_ADC\\_SConv24](#), [P2\\_Start\\_Conv](#), [P2\\_Wait\\_EOC](#)

### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
Dim i As Long  
Dim Data_1[1000] As Long 'Declaration  
  
Init:  
i=1  
P2_Set_Mux(1,0100000010b)'set MUX to input 3, gain 2  
Rem wait for multiplexer settling, here 4 µs  
P2_Sleep(400)  
P2_Start_Conv(1) 'start A/D conversion  
  
Event:  
P2_Wait_EOC(1)  
Data_1[i] = P2_Read_ADC_SConv(1)'read and start A/D converter  
Inc(i) 'increment index  
If (i=1001) Then End 'End process after 1000 values
```

**P2\_Read\_ADC\_SConv24** reads the conversion result from an ADC of the specified module and immediately starts a new conversion.

The return value has a resolution of 24 bit.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Read_ADC_SConv24(module)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>ret_val</b>	Value from the ADC (0...16777215 = $2^{24}-1$ ).	LONG

## See also

[P2\\_ADC](#), [P2\\_ADC24](#), [P2\\_Read\\_ADC](#), [P2\\_Read\\_ADC\\_SConv](#), [P2\\_Start\\_Conv](#), [P2\\_Wait\\_EOC](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim i As Long
Dim Data_1[1000] As Long 'Declaration

Init:
  i=1
  P2_Set_Mux(1,0100000010b)'set MUX to input 3, gain 2
  Rem wait for multiplexer settling, here 4 µs
  P2_Sleep(400)
  P2_Start_Conv(1)          'start A/D conversion

Event:
  P2_Wait_EOC(1)
  Data_1[i] = P2_Read_ADC_SConv24(1)'Read + start A/D converter
  Inc(i)          'increment index
  If (i=1001) Then End      'End process after 1000 values
```

## P2\_Read\_ADC\_SConv24

## P2\_SE\_Diff

**P2\_SE\_Diff** sets the operating mode single ended or differential for all analog inputs of the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_SE_Diff(module,choice)
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>choice</b>	Operating mode of analog inputs. 0: single ended. 1: differential (default).	LONG

### Notes

The operating mode single ended provides 32 inputs, in differential mode there are 16 inputs. After power-up of the device all inputs are set to differential mode.

### See also

[P2\\_ADC](#)

### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
  
Init:  
  P2_SE_Diff(1,0)           'set module address 1 to s.e.  
  
  P2_SE_Diff(2,1)           'set module address 2 to diff.
```



**P2\_Seq\_Init** initializes the sequential control of the specified module.

These settings are done: Operating mode, gain factor, channel selection and multiplexer settling time (similar for all channels).

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_Seq_Init(module, mode, gain, channels, mux_time)
```

## Parameters

**module** Selected module address (1...15). LONG

**mode** Operating mode of the sequential control: LONG  
 0: Single conversion (default), no sequential control.  
 1: Mode "single shot", single conversion cycle.  
 2: Mode "continuous", continuous conversion.  
 3: Mode "continuous max" conversion using max. speed.

**gain** Gain factor (modes 1 ... 3 only): LONG  
 0 factor = 1, voltage range -10V...+10V.  
 1 factor = 2, voltage range -5V...+5V.  
 2 factor = 4, voltage range -2.5V...+2.5V.  
 3 factor = 8, voltage range 1.25V...+1.25V.

**channels** Bit pattern to select the channels for conversion. LONG  
 Bit = 0: Don't convert.  
 Bit = 1: Do conversion.

Bit no.	31	...	7	...	2	1	0
Channel no.	32	...	8	...	3	2	1

**mux\_time** Number of time units, which sets the settling time of the sequential control: LONG  
 0: Standard waiting time (125 = 2.5µs).  
 125...2<sup>31</sup>: Waiting time in units of 20ns.

## Notes

After power-up mode 0 is active.

Modes 1...3 activate the sequential control of the module, single conversions with **P2\_ADC** are disabled then. The sequential control consecutively runs a conversion on several channels. The sequential control is always related to those channels being selected by **channels**.

The modes differ in the following items:

Mode	Kind of conversion
0 Standard:	Single conversion at one channel without sequential control, see <b>P2_ADC</b> .
1 single shot:	The sequential control is started by <b>P2_Seq_Start</b> ; it ends as soon as each of the selected channels is converted once.  The end of the sequential control is queried with <b>P2_Seq_Wait</b> and measurement values are read with <b>P2_Seq_Read</b> .

## P2\_Seq\_Init

2 continuous: The sequential control converts all selected channels for each process cycle.

The conversion is started with **P2\_Seq\_Start** as last instruction in section **Init:**. The end of conversion (for all channels) is automatically synchronized with the beginning of the next process cycle. Therefore all measurement values can—and should be—read with **Seq\_Read** at the beginning of each process cycle .

3 continuous max: The sequential control converts the selected channels continuously, providing new measurement values all the time. That is, conversion and process cycle run non-synchronously.

The conversion is started with **P2\_Seq\_Start**. Inside a process cycle **P2\_Seq\_Read** just reads the newest measurement value.

Please note for mode 2 (continuous): The synchronization happens only once and is only valid for the set cycle time (**Processdelay**). If the process timing changes, e.g. by changing the cycle time, the synchronization is lost. The consequence is, measurement values are being read too early and thus multiple, or measurement values are lost, because they are already overwritten by new values before reading.

In a channel group any module channel may be selected. The channels of a group are automatically sorted in ascending order of channel numbers, that is the sequential control converts the channel with the smallest number first.

The status and read instructions always and solely refer to the group of selected channels.



With 32-input modules the inputs must be set as single ended or differential with **P2\_SE\_Diff**.

If the internal resistance of the signal's voltage source is too great, the pre-set multiplexer settling time is too short for an accurate measurement. You can change the multiplexer settling time with the parameter **mux\_time**.

The settling time influences the accuracy of the measurement at a high rate. Shorter settling time tends to result in less accurate measurement values and longer settling time in more accurate values.

The settling time is calculated according to the following formula:

$$\text{settling time} = \text{muxtime} \cdot 20\text{ns} + \text{conversion time}$$

The values of conversion time and pre-set settling time are given in the hardware documentation of the module.

#### See also

[P2\\_ADC](#), [P2\\_Seq\\_Read](#), [P2\\_Seq\\_Read24](#), [P2\\_Seq\\_Read\\_Packed](#), [P2\\_Seq\\_Start](#), [P2\\_Seq\\_Wait](#)

#### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Define module 1
#include ADwinPro_All.Inc

Dim Data_1[16] As Long At DM_Local

Init:
  P2_SE_Diff(module,0)      'set inputs to single ended
  REM sequential control: continuous Mode, gain 1
  REM odd-numbered channels of module AIN-32
  REM standard settling time
  P2_Seq_Init(module,3,0,5555555h,0)
  REM start measuring sequence on modules 1 and 3
  P2_Seq_Start(Shift_Left(1,module-1))
  P2_Seq_Wait(module)      'wait until all selected channels
                           'are converted once

Event:
  REM read values and copy into Data_1
  P2_Seq_Read(module,16,Data_1,1)
```

## P2\_Seq\_Read

**P2\_Seq\_Read** reads a given number of values (16 Bit) from the specified module and copies them into a destination array.  
Each array element holds 1 measurement value.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Seq_Read(module, count, array[ ], array_idx)
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>count</b>	Even number (2...32) of read measurement values. An odd number is not allowed.	LONG
<b>array[ ]</b>	Destination array to store the measurement values.	ARRAY LONG FLOAT
<b>array_idx</b>	Destination start index: first array element to store a value in (1...n).	LONG

### Notes

You can only reasonably use this instruction if the sequential control of the module has been activated before with **P2\_Seq\_Init**.

The measurement values of the channel group are copied into the destination array starting ascending from the smallest channel number.

### See also

[P2\\_Seq\\_Init](#), [P2\\_Seq\\_Read24](#), [P2\\_Seq\\_Read\\_Packed](#), [P2\\_Seq\\_Start](#), [P2\\_Seq\\_Wait](#)

### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc
#Define module 1
Dim Data_1[16] As Long At DM_Local

Init:
    P2_SE_Diff(module,0)          'set inputs to single ended
    REM sequential control: continuous Mode, gain 1
    REM odd-numbered channels, standard settling time
    P2_Seq_Init(module,3,0,55555555h,0)
    P2_Seq_Start(Shift_Left(1, module-1)) 'start sequential control
    P2_Seq_Wait(module)           'wait until all selected channels
                                'are converted once

Event:
    Rem copy current values from the module into Data_1
    P2_Seq_Read(module,16,Data_1,1)
```

**P2\_Seq\_Read24** reads a given number of values (18 Bit) from the specified module and copies them into a destination array.  
Each array element holds 1 measurement value.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Seq_Read24(module, count, array[ ], array_idx)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>count</b>	Number (1...32) of read measurement values.	LONG
<b>array[ ]</b>	Destination array to store the measurement values.	ARRAY LONG FLOAT
<b>array_idx</b>	Destination start index: first array element to store a value in (1...n).	LONG

## Notes

You can only reasonably use this instruction if the sequential control of the module has been activated before with **P2\_Seq\_Init**.

The measurement values of the channel group are copied into the destination array starting ascending from the smallest channel number.

If a measurement value has a resolution less than 24 bit, the "missing" bits in the return value „fehlenden“ are filled with zeros.

As an example, the measurement value of an 18 bit ADC is located in the bits 6...23 of the return value; the measurement value is shifted to the left by 6 bits and the bits 0...5 are zeros.

Bit no.	31...24	23...6	05...00
Content	0	18 bit meas. value	0

## See also

[P2\\_Seq\\_Init](#), [P2\\_Seq\\_Read](#), [P2\\_Seq\\_Read\\_Packed](#), [P2\\_Seq\\_Start](#), [P2\\_Seq\\_Wait](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## P2\_Seq\_Read24

## Example

```
#Include ADwinPro_All.Inc
#Define module 1
Dim Data_1[16] As Long At DM_Local

Init:
    P2_SE_Diff(module,0)      'set inputs to single ended
    REM sequential control: continuous Mode, gain 1
    REM odd-numbered channels, standard settling time
    P2_Seq_Init(module,3,0,55555555h,0)
    P2_Seq_Start(Shift_Left(1, module-1))'start sequential control
    P2_Seq_Wait(module)       'wait until all selected channels
                              are converted once

Event:
    Rem Copy current values from the module into Data_1
    P2_Seq_Read24(module,16,Data_1,1)
```

**P2\_Seq\_Read\_Packed** reads an even number of value pairs (16 Bit) from the specified module and copies them into a destination array.  
Each array element holds 2 measurement values.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Seq_Read_Packed(module,count,array[],array_idx)
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>count</b>	Number of value pairs to read (1...16).	LONG
<b>array[]</b>	Destination array to store the measurement values.	ARRAY LONG FLOAT
<b>array_idx</b>	Destination start index: first array element to store a value in (1...n).	LONG

### Notes

You can only reasonably use this instruction if the sequential control of the module has been activated before with **P2\_Seq\_Init**.

The measurement values of the channel group are copied into the destination array starting ascending from the smallest channel number and in pairs. An array element contains the value of the channel with the smaller number in the lower word, the value of the higher channel number in the upper word.

### See also

[P2\\_Seq\\_Init](#), [P2\\_Seq\\_Read](#), [P2\\_Seq\\_Read24](#), [P2\\_Seq\\_Start](#), [P2\\_Seq\\_Wait](#)

### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc

Dim Data_1[8], Data_2[8] As Long At DM_Local

Init:
P2_SE_Diff(1,0)           'set module 1 inputs to single ended
P2_SE_Diff(5,0)           'set module 5 inputs to single ended
REM modules 1+5: Sequential control continuous mode, gain 1,
REM even-numbered channels (2...16), standard settling time
P2_Seq_Init(1,3,0,0AAAAh,0)
P2_Seq_Init(5,3,0,0AAAAh,0)
P2_Seq_Start(10001b)      'start sequence control on modules 1+5
P2_Seq_Wait(1)            'wait until all selected channels
                           'are converted once

Event:
REM read 16 values and copy into Data_1 and Data_2
P2_Seq_Read_Packed(1,8,Data_1,1)
P2_Seq_Read_Packed(5,8,Data_2,1)
```

## P2\_Seq\_Read\_Packed

## P2\_Seq\_Start

**P2\_Seq\_Start** starts the sequence control on all selected modules at the same time.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Seq_Start(module_pattern)
```

### Parameters

**module\_** Bit pattern to set the module addresses:  
**pattern** Bit = 0: Ignore module address.  
Bit = 1: Select module address.

LONG

Bit pattern	31:15	14	13	...	01	00
Module address	–	15	14	...	2	1

### Notes

If a module without sequence control is selected, the instruction may cause unpredictable consequences.

### See also

[P2\\_Seq\\_Init](#), [P2\\_Seq\\_Read](#), [P2\\_Seq\\_Read24](#), [P2\\_Seq\\_Read\\_Packed](#), [P2\\_Seq\\_Wait](#)

### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc
#Define module 4 'Module address

Dim Data_1[32] As Long At DM_Local
Dim i As Long

Init:
    P2_SE_Diff(module,0) 'set inputs to single ended
    REM set sequential control to single shot,
    REM gain 1, all channels,
    REM standard settling time
    P2_Seq_Init(module,1,0,0FFFFFFFh,0)
    P2_Seq_Start(Shift_Left(1,module-1)) 'start sequential control

Event:
    P2_Seq_Wait(module) 'wait for end of conversion
    P2_Seq_Read(module,32,Data_1,1) 'read 32 channels ...
    For i=1 To 32
        REM convert digit to Volt and store
        Data_1[i] = (Data_1[i]-32768)*20/65536
    Next i
    P2_Seq_Start(Shift_Left(1,module-1)) 'start next sequence
```



**P2\_Seq\_Wait** waits until the sequence control has converted and stored all channels of the channel group on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Seq_Wait(module)
```

## Parameters

**module**      Selected module address (1...15). LONG

## Notes

If sequence controls have been started on several modules at the same time (and with the same parameter values), they will end at the same time, too.

## See also

[P2\\_Seq\\_Init](#), [P2\\_Seq\\_Read](#), [P2\\_Seq\\_Read24](#), [P2\\_Seq\\_Read\\_Packed](#), [P2\\_Seq\\_Start](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
#Define module 4                    'Module address

Dim Data_1[32] As Long At DM_Local
Dim Data_2[32] As Float At DM_Local
Dim i As Long

Init:
  Processdelay=100000
  P2_SE_Diff(module,0)            'set inputs to single ended
  REM set sequential control to single shot,
  REM gain 1, all channels,
  REM settling time 0,5 µs
  P2_Seq_Init(module,3,0,0FFFFFFFh,50)
  P2_Seq_Start(Shift_Left(1,module-1)) 'start sequence control

Event:
  P2_Seq_Wait(module)            'wait for end of conversion
  P2_Seq_Read(module,32,Data_1,1) 'read 32 channels ...
  For i=1 To 32
    REM convert digit to Volt and store
    Data_2[i] = (Data_1[i]-32768)*20/65536
  Next i
  P2_Seq_Start(Shift_Left(1,module-1)) 'start next sequence
```

## P2\_Seq\_Wait

## P2\_Set\_Mux

**P2\_Set\_Mux** sets the multiplexer of the specified module to the selected input and to the selected gain.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Set_Mux(module,pattern)
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>pattern</b>	Bit pattern for multiplexer settings (see table); bits 8...9 do gain settings, bits 0...4 set the number of the input channel.	LONG

		Bit no.							
31:7	9	8	7...5	4	3	2	1	0	
no function	gain		–	Multiplexer input					
	1 = 00b		–	input 1: 00000b					
	2 = 01b			input 2: 00001b					
	4 = 10b			...					
	8 = 11b			input 32: 11111b					

### Notes

Combine the adequate bit combinations for gain and multiplexer input to find the wanted multiplexer settings.

You may set the bits of parameter **pattern** in binary format or convert them into hexadecimal or decimal format. For hex or binary formats, please note the character suffix **h** and **b**.

Please note the required multiplexer settling time (see hardware documentation). Make sure that this settling time passes at minimum between setting the multiplexer and start of conversion.

### See also

[P2\\_ADC](#), [P2\\_Start\\_Conv](#), [P2\\_Wait\\_EOC](#), [P2\\_Read\\_ADC](#)

### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim value1 As Long 'Declaration

Event:
P2_Set_Mux(1,0100000010b)'set MUX to input 3, gain 2
Rem wait for multiplexer settling, here 4 µs
P2_Sleep(400)
P2_Start_Conv(1) 'start AD conversion
P2_Wait_EOC(1) 'Wait for end of conversion
value1 = P2_Read_ADC(1) 'Read value from ADC
```

**P2\_Start\_Conv** starts the conversion on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Start_Conv(module)
```

## Parameters

**module**      Selected module address (1...15).

LONG

## Notes

- / -

## See also

[P2\\_ADC](#), [P2\\_ADC24](#), [P2\\_Read\\_ADC](#), [P2\\_Wait\\_EOC](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim value As Long      'Declaration
```

## Event:

```
P2_Set_Mux(1,0100000010b)'set MUX to input 3, gain 2
Rem wait for multiplexer settling, here 4 µs
P2_Sleep(400)
P2_Start_Conv(1)      'start AD conversion
P2_Wait_EOC(1)      'Wait for end of conversion
value = P2_Read_ADC(1) 'Read value from ADC
```

## P2\_Start\_Conv

## P2\_Wait\_EOC

**P2\_Wait\_EOC** waits for the end of conversion on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_Wait_EOC(module)
```

### Parameters

**module**      Selected module address (1...15).

LONG

### Notes

- / -

### See also

[P2\\_ADC](#), [P2\\_ADC24](#), [P2\\_Start\\_Conv](#), [P2\\_Read\\_ADC](#)

### Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
Dim value As Long      'Declaration
```

#### Event:

```
P2_Set_Mux(1,0100000010b)'set MUX to input 3, gain 2  
Rem wait for multiplexer settling, here 4 µs  
P2_Sleep(400)  
P2_Start_Conv(1)      'start AD conversion  
P2_Wait_EOC(1)      'Wait for end of conversion  
value = P2_Read_ADC(1)      'Read value from ADC
```

**P2\_Wait\_Mux** waits for the end of the multiplexer settling on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Wait_Mux(module)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
---------------	-----------------------------------	------

## Notes

If you set the multiplexer to a different channel using **P2\_Set\_Mux** it takes a defined time until the multiplexer is settled. The instruction **P2\_Wait\_Mux** waits until this moment, so immediately afterwards you can start a A/D conversion.

If the multiplexer is set to the same channel as the previous conversion, or if the multiplexer has already settled, the waiting time is skipped automatically.

## See also

[P2\\_ADC](#), [P2\\_ADC24](#), [P2\\_Set\\_Mux](#), [P2\\_Start\\_Conv](#), [P2\\_Read\\_ADC](#)

## Valid for

Aln-16/18-8B Rev. E, Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-8/18-8B Rev. E

## Example

```
#Include ADwinPro_All.Inc
#Define module 1
```

### Init:

```
P2_Seq_Init(module,0,0,0,0) 'switch off sequential control mode
P2_Set_Mux(module,0b)      'set multiplexer to input 1, gain 1
P2_Wait_Mux(module)
P2_Start_Conv(module)      'start AD conversion
Processdelay=30000         'cycle-time 0.1 ms
```

### Event:

```
P2_Set_Mux(module,0100000001b) 'set MUX to input 2, gain 2
P2_Wait_EOC(module)             'wait for end of conversion
Par_1 = P2_Read_ADC(module)     'read channel value 1 from the ADC
P2_Wait_Mux(module)             'wait for end of settling time
P2_Start_Conv(module)           'start AD conversion
P2_Set_Mux(module,0b)           'set MUX to input 1, gain 1
P2_Wait_EOC(module)             'wait for end of conversion
Par_2 = P2_Read_ADC(module)     'read channel value 2 from the ADC

P2_Wait_Mux(module)             'wait for end of settling time
P2_Start_Conv(module)           'start AD conversion
```

## P2\_Wait\_Mux

## P2\_Burst\_CRead\_Unpacked1

**P2\_Burst\_CRead\_Unpacked1** copies an amount of the last measured values of a channel from the memory of the specified module into an array.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Burst_CRead_Unpacked1(module, count, array[],

    array_idx, flowrate)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Amount of measurement values to be transferred. The amount must be divisible by 8.	LONG
<b>array[ ]</b>	Destination array, where the measurement values are transferred. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
<b>array_idx</b>	Destination start index: Array element from which measurement values are stored.	LONG
<b>flowrate</b>	Evaluation for low-priority processes only: Parameter for data throughput. 1: slow. 2: medium. 3: fast.	LONG

### Notes

The instruction be used if a continuous burst sequence was initialized with 1 channel (see **P2\_Burst\_Init**, parameters **mode**, **channels**).

The instruction reads the amount of **count** measurement values that are stored in the module memory. **count** should be lower by the factor 2 than the number of measurements (**samples**), specified in **P2\_Burst\_Init**

The instruction stores the measurement values one after the other in the elements of the destination array.

In high-priority processes the maximum data throughput is used automatically; the parameter **flowrate** must be indicated all the same.

The higher the data throughput—in a low priority process only—is selected, the more it may happen that a process of higher priority is delayed.

### See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_CRead\\_Unpacked2](#), [P2\\_Burst\\_CRead\\_Unpacked4](#), [P2\\_Burst\\_CRead\\_Unpacked8](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#)

### Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E



### Example

```
#Include ADwinPro_All.Inc
#Define module 4

Dim Data_1[1000] As Long
Dim pattern As Long

Init:
    REM Initiate cont. burst sequence for channel 1 using 20ns
    REM period duration, save 2^26 samples from address 0.
    P2_Burst_Init (module,1,0,67108864,1,010b)
    REM Start burst sequence
    pattern = Shift_Left(1,module-1) 'access single module only
    P2_Burst_Start(pattern)
    Processdelay=10000000

Event:
    REM Read last 1000 samples from channel (slowly) and store
    REM in Data_1
    P2_Burst_CRead_Unpacked1(module,1000,Data_1,1,1)
```

## P2\_Burst\_CRead\_Unpacked2

**P2\_Burst\_CRead\_Unpacked2** copies an amount of the last measurement values of 2 channels from the memory of the specified module into 2 arrays.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Burst_CRead_Unpacked2(module, count, array1[],
                        array2[], array_idx, flowrate)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Amount of measurement values per channel to be transferred. The amount must be divisible by 4.	LONG
<b>arrayx[ ]</b>	Destination arrays for the measurement values of channels 1 and 2.	ARRAY LONG
	No float type and no FIFO array allowed.	FLOAT
<b>array_idx</b>	Destination start index: Array element from which measurement values are stored.	LONG
<b>flowrate</b>	Evaluation for low-priority processes only: Parameter for data throughput. 1: slow. 2: medium. 3: fast.	LONG

### Notes

The instruction be used if a continuous burst sequence was initialized with 2 channels (see **P2\_Burst\_Init**, parameters **mode**, **channels**).

The instruction reads the amount of **count** measurement values that are stored in the module memory. **count** should be lower by the factor 2 than the number of measurements (**samples**), specified in **P2\_Burst\_Init**

The instruction stores the measurement values one after the other in the elements of the destination array.

In high-priority processes the maximum data throughput is used automatically; the parameter **flowrate** must be indicated all the same.

The higher the data throughput—in a low priority process only—is selected, the more it may happen that a process of higher priority is delayed.

### See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_CRead\\_Unpacked1](#), [P2\\_Burst\\_CRead\\_Unpacked4](#), [P2\\_Burst\\_CRead\\_Unpacked8](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#)

### Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E





### Example

```
#Include ADwinPro_All.INC
#Define module 4

Dim Data_1[1000], Data_2[1000] As Long
Dim pattern As Long

Init:
    REM Initiate cont. burst sequence for channels 1...2 using 60ns
    REM period duration, save 2^26 samples per channel starting
    REM from address 0.
    P2_Burst_Init (module,3,0,67108860,3,010b)
    REM Start burst sequence
    pattern = Shift_Left(1,module-1) 'access single module only
    P2_Burst_Start(pattern)
    P2_Set_LED(module,1)
    Processdelay=10000000

Event:
    REM Read last 1000 samples per channel (slowly) and store
    REM in Data_1 and Data_2
    P2_Burst_CRead_Unpacked2(module,1000,Data_1,Data_2,1,1)
```

## P2\_Burst\_CRead\_Unpacked4

**P2\_Burst\_CRead\_Unpacked4** copies an amount of the last measurement values of 4 channels from the memory of the specified module into 4 arrays.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Burst_CRead_Unpacked4(module, count, array1[],
    array2[], array3[], array4[], array_idx,
    flowrate)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Amount of measurement values per channel to be transferred. The amount must be divisible by 2.	LONG
<b>arrayx[ ]</b>	Destination arrays for the measurement values of channels 1...4.	ARRAY LONG
	No float type and no FIFO array allowed.	FLOAT
<b>array_idx</b>	Destination start index: Array element from which measurement values are stored.	LONG
<b>flowrate</b>	Evaluation for low-priority processes only: Parameter for data throughput. 1: slow. 2: medium. 3: fast.	LONG

### Notes

The instruction be used if a continuous burst sequence was initialized with 4 channels (see **P2\_Burst\_Init**, parameters **mode**, **channels**).

The instruction reads the number of **count** measurement values that are stored in the module memory. **count** should be lower by the factor 2 than the number of measurements (**samples**), specified in **P2\_Burst\_Init**.

The instruction stores the measurement values one after the other in the elements of the destination array.

In high-priority processes the maximum data throughput is used automatically; the parameter **flowrate** must be indicated all the same.

The higher the data throughput—in a low priority process only—is selected, the more it may happen that a process of higher priority is delayed.

### See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_CRead\\_Unpacked1](#), [P2\\_Burst\\_CRead\\_Unpacked2](#), [P2\\_Burst\\_CRead\\_Unpacked8](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#)

### Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E



## Example

```
#Include ADwinPro_All.Inc
#Define module 4

Dim Data_1[1000], Data_2[1000] As Long
Dim Data_3[1000], Data_4[1000] As Long
Dim pattern As Long

Init:
    REM Initiate cont. burst sequence for channels 1...4 using 40ns
    REM period duration, save 2^25 samples per channel starting
    REM from address 0.
    P2_Burst_Init (module,15,0,3355444,2,010b)
    REM Start burst sequence
    pattern = Shift_Left(1,module-1) 'access single module only
    P2_Burst_Start(pattern)
    Processdelay=50000000

Event:
    REM Read last 1000 samples per channel (fast) and store
    REM in Data_1 to Data_4
    P2_Burst_CRead_Unpacked4(module,1000,Data_1,Data_2,Data_3,
        Data_4,1,3)
```

## P2\_Burst\_CRead\_Unpacked8

**P2\_Burst\_CRead\_Unpacked8** copies an amount of the last measurement values of 8 channels from the memory of the specified module into 8 arrays.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Burst_CRead_Unpacked8(module, count, array1[],
    array2[], array3[], array4[], array5[], array6[],
    array7[], array8[], array_idx, flowrate)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Amount of measurement values per channel to be transferred. The amount of values must be divisible by 4.	LONG
<b>arrayx[ ]</b>	Destination arrays for the measurement values of channels 1...8.	ARRAY LONG
	No float type and no FIFO array allowed.	FLOAT
<b>array_idx</b>	Destination start index: Array element from which measurement values are stored.	LONG
<b>flowrate</b>	Evaluation for low-priority processes only: Parameter for data throughput. 1: slow. 2: medium. 3: fast.	LONG

### Notes

The instruction be used if a continuous burst sequence was initialized with 8 channels (see **P2\_Burst\_Init**, parameters **mode**, **channels**).

The instruction reads the number of **count** measurement values that are stored in the module memory. **count** should be lower by the factor 2 than the number of measurements (**samples**), specified in **P2\_Burst\_Init**.

The instruction stores the measurement values one after the other in the elements of the destination array.

In high-priority processes the maximum data throughput is used automatically; the parameter **flowrate** must be indicated all the same.

The higher the data throughput—in a low priority process only—is selected, the more it may happen that a process of higher priority is delayed.

### See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_CRead\\_Unpacked1](#), [P2\\_Burst\\_CRead\\_Unpacked2](#), [P2\\_Burst\\_CRead\\_Unpacked4](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#)

### Valid for

Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E



## Example

```
#Include ADwinPro_All.Inc
#Define module 4

Dim Data_1[1000], Data_2[1000] As Long At DM_Local
Dim Data_3[1000], Data_4[1000] As Long At DM_Local
Dim Data_5[1000], Data_6[1000] As Long At DM_Local
Dim Data_7[1000], Data_8[1000] As Long At DM_Local
Dim pattern As Long

Init:
    REM Initiate cont. burst sequence for channels 1...4 using 40ns
    REM period duration, save 2^25 samples per channel starting
    REM from address 0.
    P2_Burst_Init (module,255,100,1000000,2,010b)
    REM Start burst sequence
    pattern = Shift_Left(1,module-1) 'access single module only
    P2_Burst_Start(pattern)
    Processdelay=10000000

Event:
    REM Read last 10000 samples per channel (fast) and store
    REM in Data_1 to Data_8
    P2_Burst_CRead_Unpacked8(module,1000,Data_1,Data_2,Data_3,
        Data_4,Data_5,Data_6,Data_7,Data_8,1,3)
```

## P2\_Burst\_Init

**P2\_Burst\_Init** sets the parameters for a burst-measurement sequence on the specified module.

These are: Measurement mode (amount and numbers of measurement channels), start address in module memory, period duration of measurement sequence and number of measurements to be executed.

### Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_Burst_Init (module, channels, startadr, samples,  
              pulses, mode)
```

### Parameters

**module** Specified module address (1...15). LONG

**channels** specifies amount and numbers of measurement channels. Only given values are allowed. LONG  
In combination with the memory size the maximum amount of measurement values per channel is set:

channel ls	Channel no.	max. amount of measurement values per channel for startadr=0:
1	1	134217720 = 7FFFFFF0H
3	1...2	67108860 = 3FFFFFFCh
15	1...4	33554428 = 1FFFFFFCh
255	1...8	16777212 = 0FFFFFFCh

Alternatively the last channel may be used as time channel with the following values:

channel ls	channel no.	time ch.	max. amount of values per channel for startadr=0:
131	1	2	67108860
143	1...3	4	33554428
127	1...7	8	16777212

**startadr** Start address ( $0 \dots 268435452 = 2^{28} - 4$ ) in the module memory. Address must be divisible by 4. LONG

**samples** Amount of measurements per channel to be executed (the maximum amount is determined by **channels**). The amount must be divisible by 4; if **channels**=1 (1channel) it must be divisible by 8. LONG

**pulses** determines the period duration of a measurement sequence as number of time intervals; valid only with timer-controlled speed (see **mode**): LONG

period duration = **pulses** \* 20ns.

The value range is 1...65535; with 8 channels (**mode**=255 or 127) the range starts with 2.

The period duration is the time from the beginning of a measurement until the beginning of the next measurement.

*For module types AIn-F-x/16 only:*

period duration = **pulses** \* 10ns.

The smallest value of **pulses** depends on the parameter **mode** of **P2\_Set\_Average\_Filter**:

<b>mode</b>	<b>pulses<sub>min</sub></b>
0	25
1	67
2	154
3	313
4	645
5	1333

**mode** Bit pattern, defining the mode of burst sequence: LONG

Bit no.	03...31	02	01	00
Function	–	Type of clock speed: Bit = 0: Timer controlled ( <b>pulses</b> ). Bit = 1: Externally controlled (event input).	Operating mode of burst sequence: Bit = 0: Single. Bit = 1: Continuous.	–

## Notes

You can read the current measurement value of a channel even with **P2\_Read\_ADCF** if it is not saved, for instance for testing a trigger condition.

The time channel stores with each burst measurement the current value of the module timer. Thus, the values can be exactly allocated on a timeline e.g. with externally controlled speed. One time unit of the 16 bit-timer relates to 20ns.

The number of storable measurement values per channel depends on the given start address and the module's memory size.

With a single burst sequence the module converts and stores a fixed number of values; settings see **P2\_Burst\_Init**. As soon as all values are stored, the burst sequence stops. Values be read using **P2\_BURST\_READ\_Unpacked...**.

With a continuous burst sequence the module converts continuously with pre-defined cycle duration. The burst sequence be stopped with **P2\_Burst\_Stop** and values be read using **P2\_Burst\_CRead\_Unpacked...**. The module stores values in the allocated memory (see **P2\_Burst\_Init**) that is used as ring buffer. Therefore the youngest value will each overwrite the eldest value.

## Operating mode

## Clock speed

With timer controlled speed the clock rate of the burst sequence is set with **pulses**.

With externally controlled speed each event signal starts a burst measurement; please note the settings of **P2\_Event\_Config**. The maximum clock rate is 50MHz. Burst sequences on several modules may be synchronized using **P2\_Sync\_Mode**.

### See also

[P2\\_Burst\\_CRead\\_Unpacked1](#), [P2\\_Burst\\_CRead\\_Unpacked2](#), [P2\\_Burst\\_CRead\\_Unpacked4](#), [P2\\_Burst\\_CRead\\_Unpacked8](#), [P2\\_Set\\_Average\\_Filter](#)

[P2\\_Burst\\_Read\\_Index](#), [P2\\_Burst\\_Read\\_Unpacked1](#), [P2\\_Burst\\_Read\\_Unpacked2](#), [P2\\_Burst\\_Read\\_Unpacked4](#), [P2\\_Burst\\_Read\\_Unpacked8](#)

[P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#), [P2\\_Burst\\_Stop](#), [P2\\_Read\\_ADCF](#), [P2\\_Sync\\_Mode](#)

### Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E

### Example

```
#Include ADwinPro_All.Inc
#Define module 4
```

```
Dim Data_1[1000] As Long
Dim pattern As Long
```

#### Init:

```
REM Initiate continuous burst sequence for channel 1 using 20ns
REM period duration, save 2^26 samples from address 0.
```

```
P2_Burst_Init (module,1,0,67108864,1,010b)
```

```
REM Start burst sequence
```

```
pattern = Shift_Left(1,module-1) 'access single module only
```

```
P2_Burst_Start(pattern)
```

```
Processdelay=10000000
```

#### Event:

```
REM Read last 1000 samples from channel (slowly) and store
REM in Data_1
```

```
P2_Burst_CRead_Unpacked1(module,1000,Data_1,1,1)
```



**P2\_Burst\_Read\_Index** returns the address in the module memory, where the last measurement values have been stored.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Burst_Read_Index(module)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Address (0...268435452 = $2^{28} - 4$ ) in the module memory.. The address is divisible by 4.	LONG

## Notes

**P2\_Burst\_Read\_Index** is an elementary instruction enabling special solutions in combination with **P2\_Burst\_Read**, but requires particular accuracy and knowledge of programming. The more simple alternative is to use the instructions **P2\_BURST\_READ\_Unpacked...** or **P2\_Burst\_CRead\_Unpacked...**.

Starting from the returned address **ret\_val** the number **n** of saved values may be calculated. The start address and the number of channels are set with **P2\_Burst\_Init**:

$$n = (\text{ret\_val} - \text{startadr}) \cdot \frac{2}{\text{no. of channels}}$$

The module memory is always addressed in steps of 4 (4 times 32 bits). After the instructions **P2\_Burst\_Init** and **P2\_Burst\_Reset** the address pointer is set to the last possible address of the reserved module memory, which is **startadr + samples \* (no. of channels) - 4**.

It depends on the measurement mode which channels provide the last measurement values in the memory. More see **P2\_Burst\_Read**.

## See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_Read\\_Unpacked1](#), [P2\\_Burst\\_Read\\_Unpacked2](#), [P2\\_Burst\\_Read\\_Unpacked4](#), [P2\\_Burst\\_Read\\_Unpacked8](#), [P2\\_Burst\\_Reset](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#), [P2\\_Burst\\_Stop](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E

## P2\_Burst\_Read\_Index

### Example

```
#Include ADwinPro_All.Inc

#Define module 4
#Define samples 500000
#Define channels 4
#Define frq_Hz 5000
#Define mem_idx Par_1
#Define count Par_2
#Define overflow Par_3

Dim Data_1[samples], Data_2[samples] As Long
Dim Data_3[samples], Data_4[samples] As Long
Dim i, prev_mem_idx, start_idx As Long

LowInit:
  For i = 1 To samples
    Data_1[i] = 0 : Data_2[i] = 0 : Data_3[i] = 0 : Data_4[i] = 0
  Next i

Init:
  Processdelay = 300000000 / frq_Hz
  P2_Set_LED(module, 1) 'switch on LED
  REM Continuous burst sequence for channels 1...4 using 100ns
  REM period duration
  P2_Burst_Init(module, 15, 0, samples, 5, 2)
  P2_Burst_Start(Shift_Left(1, module - 1))
  start_idx = 1
  prev_mem_idx = 0
  overflow = 0

Event:
  REM current memory address
  mem_idx = P2_Burst_Read_Index(module)
  REM no. of new samples per channel since last cycle
  count = (mem_idx - prev_mem_idx) * 2 / channels

  If (count > 0) Then
    REM read samples from F8/14 module
    P2_Burst_Read_Unpacked4(module, count, prev_mem_idx,
      Data_1, Data_2, Data_3, Data_4, start_idx, 0)
    REM Start index for next cycle
    start_idx = start_idx + count
    REM store index of F8/14 module
    prev_mem_idx = mem_idx
  EndIf

  If (count < 0) Then
    REM No. of samples until end of DATA
    count = samples - prev_mem_idx * 2 / channels
    REM read samples from F8/14 module
    P2_Burst_Read_Unpacked4(module, count, prev_mem_idx,
      Data_1, Data_2, Data_3, Data_4, start_idx, 0)
    REM Start index for next cycle
    start_idx = 1
    REM store index of F8/14 module for next cycle
    prev_mem_idx = 0
    Inc(overflow) 'increase overflow counter
  EndIf

Finish:
  P2_Set_LED(module, 0) 'switch off LED
```

**P2\_Burst\_Read** copies 32-bit values from the memory of the specified module into a specified array.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Burst_Read(module, count, startadr, array[],
              array_idx, flowrate)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Amount of 32-bit values to be transferred The amount must be divisible by 4.	LONG
<b>startadr</b>	Start address ( $0 \dots 268435452 = 2^{28} - 4$ ) in the module memory: Address from which the measurement values are read. The address must be divisible by 4.	LONG
<b>array[]</b>	Destination array, where the measurement values are transferred. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
<b>array_idx</b>	Destination start index: Array element from which measurement values are stored.	LONG
<b>flowrate</b>	Evaluation for low-priority processes only: Parameter for data throughput. 1: slow. 2: medium. 3: fast.	LONG

## Notes

**P2\_Burst\_Read** is an elementary instruction enabling special solutions in combination with **P2\_Burst\_Read\_Index**, but requires particular accuracy and knowledge of programming. The more simple alternative is to use the instructions **P2\_BURST\_READ\_Unpacked...** or **P2\_Burst\_CRead\_Unpacked...**.

**P2\_Burst\_Read** copies the 32 bit values from the memory without changes; any 32 bit value holds 2 measurement data of 16 bit. It depends on the set number of channels (see **P2\_Burst\_Init**, parameter **channels**) which channels the measurement data are assigned to. The following tables show the assignment of 16 bit data D to channel numbers C:

address	Bits 31:16	Bits 15:00
<b>startadr</b>	C1 / D2	C1 / D1
<b>startadr+1</b>	C1 / D4	C1 / D3
<b>startadr+2</b>	C1 / D6	C1 / D5
...	...	...
No. of channels: 1		

address	Bits 31:16	Bits 15:00
<b>startadr</b>	C2 / D1	C1 / D1
<b>startadr+1</b>	C2 / D2	C1 / D2
<b>startadr+2</b>	C2 / D3	C1 / D3
...	...	...
No. of channels: 2		

## P2\_Burst\_Read

address	Bits 31:16	Bits 15:00
<a href="#">startadr</a>	C2 / D1	C1 / D1
<a href="#">startadr+1</a>	C4 / D1	C3 / D1
<a href="#">startadr+2</a>	C2 / D2	C1 / D2
<a href="#">startadr+3</a>	C4 / D2	C3 / D2
<a href="#">startadr+4</a>	C2 / D3	C1 / D3
...	...	...

No. of channels: 4

address	Bits 31:16	Bits 15:00
<a href="#">startadr</a>	C2 / D1	C1 / D1
<a href="#">startadr+1</a>	C4 / D1	C3 / D1
<a href="#">startadr+2</a>	C6 / D1	C5 / D1
<a href="#">startadr+3</a>	C8 / D1	C7 / D1
<a href="#">startadr+4</a>	C2 / D2	C1 / D2
...	...	...

No. of channels: 8

In high-priority processes the maximum data throughput is used automatically; the parameter [flowrate](#) must be indicated all the same.

The higher the data throughput—in a low priority process only—is selected, the more it may happen that a process of higher priority is delayed.

#### See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_Read\\_Index](#), [P2\\_Burst\\_Read\\_Unpacked1](#), [P2\\_Burst\\_Read\\_Unpacked2](#), [P2\\_Burst\\_Read\\_Unpacked4](#), [P2\\_Burst\\_Read\\_Unpacked8](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#), [P2\\_Burst\\_Stop](#), [P2\\_Read\\_ADCF](#), [P2\\_Set\\_Average\\_Filter](#)

#### Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E

#### Example

see [P2\\_Burst\\_Read\\_Index](#)



**P2\_Burst\_Read\_Unpacked1** copies the measurement values of a channel into a specified array.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Burst_Read_Unpacked1(module, count, startadr,
    array[], array_idx, flowrate)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Amount of measurement values to be transferred. The amount must be divisible by 8.	LONG
<b>startadr</b>	Start address ( $0 \dots 268435452 = 2^{28} - 4$ ) in the module memory: Address from which the measurement values are read. The address must be divisible by 4.	LONG
<b>array[]</b>	Destination array, where the measurement values are transferred. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
<b>array_idx</b>	Destination start index: Array element from which measurement values are stored.	LONG
<b>flowrate</b>	Evaluation for low-priority processes only: Parameter for data throughput. 1: slow. 2: medium. 3: fast.	LONG

## Notes

The instruction be used if the burst sequence was initialized with 1 channel (see **P2\_Burst\_Init**, parameter **channels**).

The instruction stores the measurement values one after the other in the elements of the destination array.

In high-priority processes the maximum data throughput is used automatically; the parameter **flowrate** must be indicated all the same.

The higher the data throughput—in a low priority process only—is selected, the more it may happen that a process of higher priority is delayed.

## See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_Read\\_Unpacked2](#), [P2\\_Burst\\_Read\\_Unpacked4](#), [P2\\_Burst\\_Read\\_Unpacked8](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E

## P2\_Burst\_Read\_Unpacked1



### Example

See also examples for [Continuous signal conversion](#) on page 283.

```
#Include ADwinPro_All.Inc
#Define module 1

Dim Data_1[1000] As Long
Dim state As Long
Dim rest As Long
Dim pattern As Long

Init:
    REM Initiate single burst sequence for channel 1 using 20ns
    REM period duration, save 1000 samples from address 0.
    P2_Burst_Init (module,1,0,1000,1,0)
    REM Start burst sequence
    pattern = Shift_Left(1,module-1) 'access single module only
    P2_Burst_Start(pattern)
    Processdelay=10000000
    REM State: Burst_sequence is running
    state=0

Event:
    REM Get number of samples still to do
    rest=P2_Burst_Status(module)
    REM All samples done: change state
    If (rest=0) Then state=1
    If (state=1) Then
        REM All samples done: Read 1000 samples (fast) and store
        REM in Data_1
        P2_Burst_Read_Unpacked1(module,1000,0,Data_1,1,3)
        REM Start next burst sequence
        state=0
        P2_Burst_Reset(pattern)
        P2_Burst_Start(pattern)
    EndIf
```

**P2\_Burst\_Read\_Unpacked2** copies the measurement values of 2 channels from the memory of the specified module into 2 arrays.

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_Burst_Read_Unpacked2(module, count, startadr,  
    array1[], array2[], array_idx, flowrate)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Amount of measurement values to be transferred. The amount must be divisible by 4.	LONG
<b>startadr</b>	Start address ( $0 \dots 268435452 = 2^{28} - 4$ ) in the module memory: Address from which the measurement values are read. The address must be divisible by 4.	LONG
<b>arrayx[]</b>	Destination arrays for the measurement values of channels 1 and 2. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
<b>array_idx</b>	Destination start index: Array element from which measurement values are stored.	LONG
<b>flowrate</b>	Evaluation for low-priority processes only: Parameter for data throughput. 1: slow. 2: medium. 3: fast.	LONG

## Notes

The instruction be used if the burst sequence was initialized with 2 channels (see **P2\_Burst\_Init**, parameter **channels**).

The instruction stores the measurement values one after the other in the elements of the destination array: Channel 1 in **array1**, channel 2 in **array2**.

In high-priority processes the maximum data throughput is used automatically; the parameter **flowrate** must be indicated all the same.

The higher the data throughput—in a low priority process only—is selected, the more it may happen that a process of higher priority is delayed.

## See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_Read\\_Unpacked1](#), [P2\\_Burst\\_Read\\_Unpacked4](#), [P2\\_Burst\\_Read\\_Unpacked8](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E

## P2\_Burst\_Read\_Unpacked2



### Example

See also examples for [Continuous signal conversion](#) on page 283.

```
#Include ADwinPro_All.Inc
#Define module 4

Dim Data_1[1000], Data_2[1000] As Long
Dim state As Long
Dim rest As Long
Dim pattern As Long

Init:
    REM Initiate single burst sequence for channels 1 and 2 using
    REM 40ns period duration, save 1000 samples from address 0.
    P2_Burst_Init (module,3,0,1000,2,0)
    REM Start burst sequence
    pattern = Shift_Left(1,module-1) 'access single module only
    P2_Burst_Start(pattern)
    Processdelay=10000000
    REM State: Burst_sequence is running
    state=0

Event:
    REM Get number of samples still to do
    rest=P2_Burst_Status(module)
    If (rest=0) Then state=1
    If (state=1) Then
        REM All samples done: Read 1000 samples for each channel (fast)
        REM and store in Data_1 and Data_2
        P2_Burst_Read_Unpacked2(module,1000,0,Data_1,Data_2,1,3)
        REM Start next burst sequence
        state=0
        P2_Burst_Reset(pattern)
        P2_Burst_Start(pattern)
    EndIf
```



**P2\_Burst\_Read\_Unpacked4** copies the measurement values of 4 channels from the memory of the specified module into 4 arrays.

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_Burst_Read_Unpacked4(module, count, startadr,  
    array1[], array2[], array3[], array4[],  
    array_idx, flowrate)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Amount of measurement values to be transferred. The amount must be divisible by 2.	LONG
<b>startadr</b>	Start address ( $0 \dots 268435452 = 2^{28} - 4$ ) in the module memory: Address from which the measurement values are read. The address must be divisible by 4.	LONG
<b>arrayx[]</b>	Destination arrays for the measurement values of channels 1...4. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
<b>array_idx</b>	Destination start index: Array element from which measurement values are stored.	LONG
<b>flowrate</b>	Evaluation for low-priority processes only: Parameter for data throughput. 1: slow. 2: medium. 3: fast.	LONG

## Notes

The instruction be used if the burst sequence was initialized with 4 channels (see **P2\_Burst\_Init**, parameter **channels**).

The instruction stores the measurement values one after the other in the elements of the destination array: Channel 1 in **array1**, channel 2 in **array2** etc.

In high-priority processes the maximum data throughput is used automatically; the parameter **flowrate** must be indicated all the same.

The higher the data throughput—in a low priority process only—is selected, the more it may happen that a process of higher priority is delayed.

## See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_Read\\_Unpacked1](#), [P2\\_Burst\\_Read\\_Unpacked2](#), [P2\\_Burst\\_Read\\_Unpacked8](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E

## P2\_Burst\_Read\_Unpacked4



### Example

See also examples for [Continuous signal conversion](#) on page 283.

```
#Include ADwinPro_All.Inc
#Define module 4

Dim Data_1[1000], Data_2[1000] As Long
Dim Data_3[1000], Data_4[1000] As Long
Dim state As Long
Dim rest As Long
Dim pattern As Long

Init:
    REM Initiate single burst sequence for channels 1...4 using 40ns
    REM period duration, save 3000 samples per channel from addr. 0
    P2_Burst_Init (module,15,0,3000,2,0)
    REM Start burst sequence
    pattern = Shift_Left(1,module-1) 'access single module only
    P2_Burst_Start(pattern)
    Processdelay=10000000
    REM State: Burst_sequence is running
    state=0

Event:
    REM Get number of samples still to do
    rest=P2_Burst_Status(module)
    If (rest=0) Then state=1
    If (state=1) Then
        REM All samples done: Read 3000 samples (fast) per channel and
        REM store in Data_1 to Data_4
        P2_Burst_Read_Unpacked4(module,1000,0,Data_1,Data_2,Data_3,
            Data_4,1,3)
        REM Start next burst sequence
        state=0
        P2_Burst_Reset(pattern)
        P2_Burst_Start(pattern)
    EndIf
```

**P2\_Burst\_Read\_Unpacked8** copies the measurement values of 8 channels from the memory of the specified module into 8 arrays.

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_Burst_Read_Unpacked8(module, count, startadr,
    array1[], array2[], array3[], array4[], array5[],
    array6[], array7[], array8[], array_idx,
    flowrate)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Amount of measurement values to be transferred.	LONG
<b>startadr</b>	Start address ( $0 \dots 268435452 = 2^{28} - 4$ ) in the module memory: Address from which the measurement values are read. The address must be divisible by 4.	LONG
<b>arrayx[]</b>	Destination arrays for the measurement values of channels 1...8. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
<b>array_idx</b>	Destination start index: Array element from which measurement values are stored.	LONG
<b>flowrate</b>	Evaluation for low-priority processes only: Parameter for data throughput. 1: slow. 2: medium. 3: fast.	LONG

## Notes

The instruction be used if the burst sequence was initialized with 1 channel (see **P2\_Burst\_Init**, parameter **channels**).

The instruction stores the measurement values one after the other in the elements of the destination array: Channel 1 in **array1**, channel 2 in **array2** etc.

In high-priority processes the maximum data throughput is used automatically; the parameter **flowrate** must be indicated all the same.

The higher the data throughput—in a low priority process only—is selected, the more it may happen that a process of higher priority is delayed.

## See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_Read\\_Unpacked1](#), [P2\\_Burst\\_Read\\_Unpacked2](#), [P2\\_Burst\\_Read\\_Unpacked4](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Status](#)

## Valid for

Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E

## P2\_Burst\_Read\_Unpacked8



### Example

See also examples for [Continuous signal conversion](#) on page 283.

```
#Include ADwinPro_All.Inc
#Define module 4

Dim Data_1[1000], Data_2[1000] As Long
Dim Data_3[1000], Data_4[1000] As Long
Dim Data_5[1000], Data_6[1000] As Long
Dim Data_7[1000], Data_8[1000] As Long
Dim state As Long
Dim rest As Long
Dim pattern As Long

Init:
    REM Initiate single burst sequence for channels 1...8 using 40ns
    REM period duration, save 1000 samples from address 0.
    P2_Burst_Init (module,255,0,1000,2,0)
    REM Start burst sequence
    pattern = Shift_Left(1,module-1) 'access single module only
    P2_Burst_Start(pattern)
    Processdelay=10000000
    REM State: Burst_sequence is running
    state=0

Event:
    REM Get number of samples still to do
    rest=P2_Burst_Status(module)
    If (rest=0) Then state=1
    If (state=1) Then
        REM All samples done: Read 1000 samples (fast) per channel and
        REM store in Data_1 to Data_4
        P2_Burst_Read_Unpacked4(module,1000,0,Data_1,Data_2,Data_3,
            Data_4,1,3)
        REM Start next burst sequence
        state=0
        P2_Burst_Reset(pattern)
        P2_Burst_Start(pattern)
    EndIf
```

**P2\_Burst\_Reset** resets the data pointer of burst sequences on all specified modules.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Burst_Reset(module_pattern)
```

## Parameter

**module\_** Bit pattern to access the module addresses: LONG  
**pattern** Bit = 0: Ignore module.  
 Bit = 1: Access module.

Bit no.	31:15	14	13	...	01	00
Module address	–	15	14	...	2	1

## Notes

The instruction addresses all set modules at the same time. If the instruction is not valid for a set module unexpected results may occur.

The data pointer refers to the address in the module memory where the previous values have been stored. Resetting the data pointer will have the next values stored at the start address set by **P2\_Burst\_Init**. The data pointer may be read with **P2\_Burst\_Read\_Index**.

## See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_Read\\_Index](#), [P2\\_Burst\\_CRead\\_Unpacked1](#), [P2\\_Burst\\_CRead\\_Unpacked2](#), [P2\\_Burst\\_CRead\\_Unpacked4](#), [P2\\_Burst\\_CRead\\_Unpacked8](#), [P2\\_Burst\\_Read](#), [P2\\_Burst\\_Stop](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E

## P2\_Burst\_Reset



### Example

```
#Include ADwinPro_All.Inc
#Define module 4

Dim Data_1[1000] As Long
Dim state As Long
Dim remaining As Long
Dim pattern As Long

Init:
    REM Initiate single burst sequence for channel 1 using 20ns
    REM period duration, save 1000 samples from address 0.
    P2_Burst_Init (module,1,0,1000,1,0)
    REM start burst sequence
    pattern = Shift_Left(1,module-1) 'access module 4 only
    P2_Burst_Start(pattern)
    Processdelay=10000000
    state=0

Event:
    REM get number of remaining burst measurements
    remaining = P2_Burst_Status(module)
    REM all measurements are done: change status
    If (remaining = 0) Then state = 1
    If (state = 1) Then
        REM all burst measurements are done: read 1000 values fast
        REM and store into Data_1
        P2_Burst_Read_Unpacked1(module,1000,0,Data_1,1,3)
        REM start next burst sequence
        state=0
        P2_Burst_Reset(pattern)
        P2_Burst_Start(pattern)
    EndIf
```

**P2\_Burst\_Start** starts the burst measurement sequence on all specified modules at the same time.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Burst_Start(module_pattern)
```

## Parameters

**module\_** Bit pattern to set the module addresses: LONG  
**pattern** Bit = 0: Ignore module address.  
 Bit = 1: Select module address.

Bit pattern	31:15	14	13	...	01	00
Module address	–	15	14	...	2	1

## Notes

The instruction addresses all set modules at the same time. If the instruction is not valid for a set module unexpected results may occur.

## See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_Read\\_Index](#), [P2\\_Burst\\_CRead\\_Unpacked1](#), [P2\\_Burst\\_CRead\\_Unpacked2](#), [P2\\_Burst\\_CRead\\_Unpacked4](#), [P2\\_Burst\\_CRead\\_Unpacked8](#), [P2\\_Burst\\_Status](#), [P2\\_Burst\\_Stop](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E

## Example

```
#Include ADwinPro_All.Inc
#Define module 4
```

```
Dim Data_1[1000] As Long
Dim pattern As Long
```

### Init:

```
REM Initiate cont. burst sequence for channel 1 using 20ns
REM period duration, save 2^26 samples from address 0.
P2_Burst_Init (module,1,0,67108864,1,010b)
REM Start burst sequence
pattern = Shift_Left(1,module-1) 'one module only
P2_Burst_Start(pattern)
Processdelay=10000000
```

### Event:

```
REM Read previous 1000 samples from channel (slowly) and store
REM in Data_1
P2_Burst_CRead_Unpacked1(module,1000,Data_1,1,1)
```

## P2\_Burst\_Start



## P2\_Burst\_Status

**P2\_Burst\_Status** determines the number of burst measurements which are still to execute on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
ret_val = P2_Burst_Status(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Number of measurements which are to execute.	LONG

### Notes

The instruction be used for a single burst sequence only (see **P2\_Burst\_Init**).

If a burst measurement sequence is already finished, the function returns 0 (zero).

### See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_Read\\_Index](#), [P2\\_Burst\\_Read\\_Unpacked1](#), [P2\\_Burst\\_Read\\_Unpacked2](#), [P2\\_Burst\\_Read\\_Unpacked4](#), [P2\\_Burst\\_Read\\_Unpacked8](#), [P2\\_Burst\\_Start](#), [P2\\_Burst\\_Stop](#), [P2\\_Read\\_ADCF](#)

### Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E



## Example

```
#Include ADwinPro_All.Inc
#Define module 1

Dim Data_1[1000] As Long
Dim state As Long
Dim rest As Long
Dim pattern As Long

Init:
  REM Initiate single burst sequence for channel 1 using 20ns
  REM period duration, save 1000 samples from address 0.
  P2_Burst_Init (module,1,0,1000,1,0)
  REM Start burst sequence
  pattern = Shift_Left(1,module-1) 'one module only
  P2_Burst_Start(pattern)
  Processdelay=10000000
  REM State: Burst_sequence is running
  state=0

Event:
  REM Get number of samples still to do
  rest=P2_Burst_Status(module)
  REM All samples done: change state
  If (rest=0) Then state=1
  If (state=1) Then
    REM All samples done: Read 1000 samples (fast) and store
    REM in Data_1
    P2_Burst_Read_Unpacked1(module,1000,0,Data_1,1,3)
    REM Start next burst sequence
    state=0
    P2_Burst_Reset(pattern)
    P2_Burst_Start(pattern)
  EndIf
```

## P2\_Burst\_Stop

**P2\_Burst\_Stop** stops a running burst-measurement sequence on all specified modules at the same time.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Burst_Stop(module_pattern)
```

### Parameters

**module\_** Bit pattern for accessing the module addresses: LONG  
**pattern** Bit = 0: ignore module.  
 Bit = 1: access module.

Bit no.	31:15	14	13	...	01	00
Module address	–	15	14	...	2	1

### Notes

An internal data pointer refers to the address in the module memory where the previous values have been stored. The data pointer may be read with **P2\_Burst\_Read\_Index**.

A burst sequence being stopped can be resumed with **Burst\_Start**.

**P2\_Burst\_Reset** resets the data pointer to the start address set by **P2\_Burst\_Init**. Thus, new values will overwrite previously saved values.

### See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_CRead\\_Unpacked1](#), [P2\\_Burst\\_Read\\_Unpacked1](#), [P2\\_Burst\\_Read](#), [P2\\_Burst\\_Status](#)

### Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E

## Example

```
#Include ADwinPro_All.Inc
#Define module 4

Dim Data_1[1000] As Long
Dim i As Long
Dim pattern As Long

Init:
    REM Initiate cont. burst sequence for channel 1 using 20ns
    REM period duration, save 2^26 samples from address 0.
    P2_Burst_Init (module,1,0,67108864,1,0)
    REM Initiate single burst sequence for channels 1...8 using 40ns
    REM period duration, save 1000 samples from address 0.
    P2_Burst_Init (module,255,0,1000,2,0)
    REM Start burst sequence
    pattern = Shift_Left(1,module-1) 'one module only
    P2_Burst_Start(pattern)
    Processdelay=10000000

Event:
    REM Read last 1000 samples from channel (slowly) and store
    REM in Data_1
    P2_Burst_CRead_Unpacked1(module,1000,Data_1,1,1)
    REM Abort Burst sequence, if limit is exceeded
    For i = 1 To 1000
        If (Data_1[i]>5) Then
            P2_Burst_Stop(pattern)
        EndIf
    Next
```

## P2\_Set\_Average\_Filter

**P2\_Set\_Average\_Filter** determines if the module calculates an average and of how many values the average is calculated.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_Set_Average_Filter(module, mode)
```

### Parameters

module	Specified module address (1...15).	LONG
mode	Filter mode (0...5): 0: Filter off = Original measurement values. 1: Average over $2^1 = 2$ values. 2: Average over $2^2 = 4$ values. 3: Average over $2^3 = 8$ values. 4: Average over $2^4 = 16$ values. 5: Average over $2^5 = 32$ values.	LONG

### Notes

The filter mode applies likewise for single value measurements and burst-measurements.

The moving average is always calculated from the previously converted measurement values. Since converting continuously the modules Aln-F-x/14 provide a moving average value.

### See also

[P2\\_Burst\\_Init](#), [P2\\_Burst\\_CRead\\_Unpacked1](#), [P2\\_Burst\\_Read\\_Unpacked1](#), [P2\\_Read\\_ADCF](#)

### Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
  
Init:  
    REM Initiate filter to average over 2 samples  
    P2_Set_Average_Filter(1,1)
```

**P2\_ADCF** executes a complete measurement on a Fast-ADC. The return value has a resolution of 16 bit.

## Syntax

```
#Include ADwinPro_All.Inc

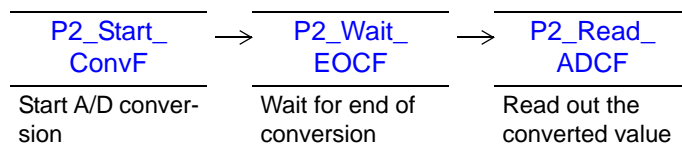
ret_val = P2_ADCF(module, input_no)
```

## Parameters

module	Specified module address (1...15).	LONG
input_no	Number of the analog input (1...4 or 1...8).	LONG
ret_val	Result of the conversion (0...65535).	LONG

## Notes

The function **P2\_ADCF** is characterized by a sequence of several instructions:



## See also

[P2\\_ADCF24](#), [P2\\_ADCF\\_Mode](#), [P2\\_ADCF\\_Read\\_Limit](#), [P2\\_ADCF\\_Set\\_Limit](#), [P2\\_Read\\_ADCF](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim value As Long
```

### Event:

```
value = P2_ADCF(1, 4)      'measure 16Bit value at analog input 4
```

## P2\_ADCF

## P2\_ADCF24

**P2\_ADCF24** executes a complete measurement on a Fast-ADC. The return value has a resolution of 24 bit.

### Syntax

```
#Include ADwinPro_All.Inc

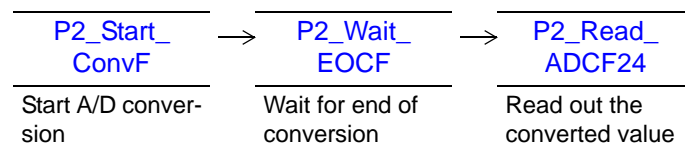
ret_val = P2_ADCF24(module, input_no)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>input_no</b>	Number of the analog input (1...4 or 1...8).	LONG
<b>ret_val</b>	Result of the conversion (0...16777215 = $2^{24}-1$ ).	LONG

### Notes

The function **P2\_ADCF24** is characterized by a sequence of several instructions:



### See also

[P2\\_ADCF](#), [P2\\_ADCF\\_Mode](#), [P2\\_ADCF\\_Read\\_Limit](#), [P2\\_ADCF\\_Set\\_Limit](#)

### Valid for

Aln-F-4/18 Rev. E, Aln-F-8/18 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim value As Long
```

#### Event:

```
REM Measure 24 bit value at analog input 4
value = P2_ADCF24(1, 4)
```

**P2\_ADCF\_Mode** sets the working mode for all channels of the selected modules.

## Syntax

```
#Include ADwinPro_All.Inc

P2_ADCF_Mode(module_pattern, mode)
```

## Parameters

**module** Bit pattern to set the module addresses: LONG  
 Bit = 0: Ignore module address.  
 Bit = 1: Select module address.

Bit pattern	31:15	14	13	...	01	00
Module address	–	15	14	...	2	1

**mode** Working mode of the module: LONG

mode	Mode
0	Standard mode (default)
1	Timer Mode
3	Timer Mode with Multiplex option <sup>a</sup>
4	Event Mode
6	Event Mode with Multiplex option <sup>a</sup>

a. not available for Aln-F-4/16 and Aln-F-8/16

## Notes

The instruction addresses all selected modules at the same time. If the instruction is not valid for a selected module unexpected results may occur.

In standard mode, the processor module starts each conversion for each channel separately, e.g. using **P2\_Start\_Conv**.

In timer mode, the module converts all channels independently and cyclic. Thus, the processor module is disburdened, furthermore only reading and processing the already converted values in the process. Each conversion on the module runs in synchrony to the **Processdelay** of the process.

The timer mode can be enabled in the **Init:** section only; the instruction should be placed at the section end.

The processor module should read the converted value in the **Event:** section first.

The following describes the action in detail:

**P2\_ADCF\_Mode** passes the currently set **Processdelay** of the process to the module. A certain time later the module independently starts conversion on all channels. The on-module timer periodically restarts the conversion and – using the passed **Processdelay** – in synchrony to the process cycle; the maximum conversion rate is given in the module's hardware description.

In timer mode the end of conversion is regularly being reached, when the processor module starts its process cycle. If the processor reads the value somewhat later–e.g. because the process cycle starts retarded or the read instruction is not first of the process cycle–the next conversion may be starting already or even be completed. Thus, the processor module may omit single values or read them more than once.

## P2\_ADCF\_Mode



### Standard Mode

### Timer Mode



#### Timer Mode with Multiplex option

Timer mode should be used in combination with a single high priority process only.

In timer mode with multiplex option the module runs with double speed than in normal timer mode, but can use only half of the channels. The processor module reads and processes a pair of converted values in each process cycle.

The measurement values can be read in pairs only, using the instructions `P2_Read_ADCF32`, `P2_Read_ADCF4_Packed`, `P2_Read_ADCF8_Packed`. The prior of both values is returned in the upper word, the subsequent value in the lower word.

Each analog signal must be connected to a pair of inputs: 1+2, 3+4, 5+6, 7+8; other pairing combinations are not allowed.

The **Processdelay** of the process must be even to synchronously clock the conversions.

#### Event Mode

In Event Mode each Event signal at the module's event input starts a conversion on all channels.

If the event input of the module is enabled with `P2_Event_Enable`, the module will send an event signal to the processor module. The event signal starts the (externally controlled) process cycle at the moment, when the end of conversion is reached. Thus, the processor module is disburdened, since it will only read and process the already converted values.

#### Event Mode with Multiplex option

In Event Mode with Multiplex option the module can run with double speed than in normal event mode, but can use only half of the channels.

Each analog signal must be connected to a pair of inputs: 1+2, 3+4, 5+6, 7+8; other pairing combinations are not allowed.

If the event input of the module is enabled with `P2_Event_Enable`, the module will send an event signal to the processor module with the end of every second conversion.

The processor module must read a pair of values in every process cycle; suitable instructions are `P2_Read_ADCF32`, `P2_Read_ADCF4_Packed`, `P2_Read_ADCF8_Packed`. The prior of both values is returned in the upper word, the subsequent value in the lower word.



There is an event input only on modules with Sub-D plugs.

#### See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_ADCF\\_Read\\_Limit](#), [P2\\_ADCF\\_Set\\_Limit](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#), [P2\\_Read\\_ADCF](#), [P2\\_Read\\_ADCF32](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#), [P2\\_Read\\_ADCF4\\_24B](#), [P2\\_Read\\_ADCF8\\_24B](#)

#### Valid for

Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E



### Example

```
#Include ADwinPro_All.Inc
Dim value[4] As Long

Init:
  Rem ...
  P2_ADCF_Mode(1,1)           'Switch on timer mode
                               'Last instruction of INIT section!

Event:
  P2_Read_ADCF4(1, value, 1) 'read values of ADC 1-4
  REM process values
```

## P2\_ADCF\_Read\_Limit

**P2\_ADCF\_Read\_Limit** reads the limit-overflow and -underrun flags of all F-ADCs on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_ADCF_Read_Limit(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Bit pattern representing the limit-overflow and -underrun flags.	LONG

Bit No.	31:24	23	22	21	20	19	18	17	16
overflow of upper limit									
Channel no.	—	8	7	6	5	4	3	2	1

---

Bit No.	15:08	7	6	5	4	3	2	1	0
underrun of lower limit									
Channel no.	—	8	7	6	5	4	3	2	1

### Notes

The limits are set with **P2\_ADCF\_Set\_Limit**.

Reading the flags resets all flags to zero.

We recommend to read the flags in the **Init:** section once, as to ensure all flags be reset. This is more important with an externally controlled process.

### See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_ADCF\\_Mode](#), [P2\\_ADCF\\_Set\\_Limit](#)

### Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim flags As Long

Init:
    P2_ADCF_Set_Limit(1, 2, 32768,256) 'set limits for channel 2
    flags = P2_ADCF_Read_Limit(1) 'read and reset flags

Event:
    flags = P2_ADCF_Read_Limit(1) 'read and reset flags
    If (flags And 10b = 10b) Then
        REM limit-overflow
        Rem ...
    EndIf
    If (flags And 2000h = 2000h) Then
        REM limit-underrun
        Rem ...
    EndIf
```

**P2\_ADCF\_Set\_Limit** sets the upper and lower limit for one F-ADC of the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_ADCF_Set_Limit(module, input_no, high, low)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>input_no</b>	Number of the analog input (1...4 or 1...8).	LONG
<b>high</b>	Upper limit (0...65535) of the channel. Default: 65535.	LONG
<b>low</b>	Lower limit (0...65535) of the channel. Default: 0.	LONG

## Notes

This instruction will run as expected only, if the module does not run in standard working mode (see **P2\_ADCF\_Mode**).

If a converted value exceeds the upper limit, the channel's flag is set. **P2\_ADC\_Read\_Limit** reads and thus resets the flags.

The same way a channel's flag is set for a converted value falling below the lower limit.

A limit-overflow or -underrun does not trigger an event signal.

## See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_ADCF\\_Mode](#), [P2\\_ADCF\\_Read\\_Limit](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim flags As Long

Init:
P2_ADCF_Set_Limit(1, 2, 32768, 256) 'Set limits for channel 2
flags = P2_ADCF_Read_Limit(1) 'read and reset flags

Event:
flags = P2_ADCF_Read_Limit(1) 'read and reset flags
If (flags And 10b = 10b) Then
    REM limit-overflow
    Rem ...
EndIf
If (flags And 20000h = 20000h) Then
    REM limit-underrun
    Rem ...
EndIf
```

## P2\_ADCF\_Set\_Limit

## P2\_ADCF\_Reset\_Min\_Max

**P2\_ADCF\_Reset\_Min\_Max** resets the minimum and maximum values of selected channels of the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_ADCF_Reset_Min_Max(module, channel_pattern)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>channel_pattern</b>	Bit pattern to select channels, where extreme minimum and maximum values are to be reset.	LONG

Bit No.	15:08	7	6	5	4	3	2	1	0
F-ADC No.	–	8	7	6	5	4	3	2	1

### Notes

The maximum values are reset to zero, the minimum values to **0FFFFh**.

### See also

[P2\\_ADCF\\_Read\\_Min\\_Max4](#), [P2\\_ADCF\\_Read\\_Min\\_Max8](#), [P2\\_ADCF\\_Read\\_Limit](#), [P2\\_ADCF\\_Set\\_Limit](#)

### Valid for

Aln-F-4/16 Rev. E, Aln-F-8/16 Rev. E

### Example

see [P2\\_ADCF\\_Read\\_Min\\_Max8](#)

**P2\_ADCF\_Read\_Min\_Max4** returns the minimum and maximum values of 4 F-ADC of the specified module in an array.

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_ADCF_Read_Min_Max4(module, array[], array_index)
```

## Parameters

<code>module</code>	Specified module address (1...15).	LONG
<code>array[]</code>	Destination array, where extremal values are stored. The array must have at least <code>array_index</code> + 7 elements.	ARRAY LONG
<code>array_index</code>	Index of the destination array element, where the first extremal value is stored.	LONG

## Notes

Extremal values are not reset by reading. For reset, use the instruction **P2\_ADCF\_Reset\_Min\_Max**.

The extremal values are stored in `array[]` in the following order (with `array_index = n`):

Array element	value, channel
<code>array[n]</code>	Min. channel 1
<code>array[n+1]</code>	Max. channel 1
<code>array[n+2]</code>	Min. channel 2
<code>array[n+3]</code>	Max. channel 2
<code>array[n+4]</code>	Min. channel 3
<code>array[n+5]</code>	Max. channel 3
<code>array[n+6]</code>	Min. channel 4
<code>array[n+7]</code>	Max. channel 4

## See also

[P2\\_ADCF\\_Read\\_Min\\_Max8](#), [P2\\_ADCF\\_Reset\\_Min\\_Max](#), [P2\\_ADCF\\_Read\\_Limit](#), [P2\\_ADCF\\_Set\\_Limit](#)

## Valid for

Aln-F-4/16 Rev. E, Aln-F-8/16 Rev. E

## P2\_ADCF\_Read\_Min\_Max4

**Example**

```
#Include ADwinPro_All.Inc
Dim Data_10[8] As Long
Dim i As Long

Init:
    P2_ADCF_Reset_Min_Max(1,1111b)'reset all 4 F-ADC

Event:
    Rem read high and low values of F-ADC 1...4
    P2_ADCF_Read_Min_Max4(1,Data_10,1)
    For i = 1 To 8 Step 2
        If (Data_10[i] < 2500) Then
            Rem minimum is below limit
            Rem ...
            P2_ADCF_Reset_Min_Max(1,1111b)'reset all 4 F-ADC
        EndIf

        If (Data_10[i+1] > 50000) Then
            Rem value is above limit
            Rem ...
        EndIf
    Next i
```

**P2\_ADCF\_Read\_Min\_Max8** gibt die Minimal- und Maximalwerte von 8 F-ADC des angegebenen Moduls in einem Feld zurück.

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_ADCF_Read_Min_Max8(module, array[], array_index)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[ ]</b>	Destination array, where extremal values are stored. The array must have at least <b>array_index</b> + 15 elements.	ARRAY LONG
<b>array_index</b>	Index of the destination array element, where the first extremal value is stored.	LONG

## Notes

Extremal values are not reset by reading. For reset, use the instruction **P2\_ADCF\_Reset\_Min\_Max**.

The extremal values are stored in **array[ ]** in the following order (with **array\_index = n**):

Array element	value, channel
<b>array[n]</b>	Min. channel 1
<b>array[n+1]</b>	Max. channel 1
<b>array[n+2]</b>	Min. channel 2
<b>array[n+3]</b>	Max. channel 2
<b>array[n+4]</b>	Min. channel 3
<b>array[n+5]</b>	Max. channel 3
<b>array[n+6]</b>	Min. channel 4
<b>array[n+7]</b>	Max. channel 4

Array element	value, channel
<b>array[n+8]</b>	Min. channel 5
<b>array[n+9]</b>	Max. channel 5
<b>array[n+10]</b>	Min. channel 6
<b>array[n+11]</b>	Max. channel 6
<b>array[n+12]</b>	Min. channel 7
<b>array[n+13]</b>	Max. channel 7
<b>array[n+14]</b>	Min. channel 8
<b>array[n+15]</b>	Max. channel 8

## See also

[P2\\_ADCF\\_Read\\_Min\\_Max4](#), [P2\\_ADCF\\_Reset\\_Min\\_Max](#), [P2\\_ADCF\\_Read\\_Limit](#), [P2\\_ADCF\\_Set\\_Limit](#)

## Valid for

Aln-F-4/16 Rev. E, Aln-F-8/16 Rev. E

## P2\_ADCF\_Read\_Min\_Max8

#### Example

```
#Include ADwinPro_All.Inc
Dim Data_4[16] As Long
Dim i As Long

Init:
    P2_ADCF_Reset_Min_Max(1,11111111b)'reset all 8 F-ADC

Event:
    Rem read high and low values of F-ADC 1...8
    P2_ADCF_Read_Min_Max8(1,Data_4,1)
    For i = 1 To 16 Step 2
        If (Data_4[i] < 2500) Then
            Rem minimum is below limit
            Rem ...
            P2_ADCF_Reset_Min_Max(1,11111111b)'reset all 8 F-ADC
        EndIf

        If (Data_4[i+1] > 50000) Then
            Rem value is above limit
            Rem ...
        EndIf
    Next i
```



**P2\_Read\_ADCF** reads out the conversion result from an F-ADC of the specified module. The return value has a resolution of 16 bit.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Read_ADCF(module, adc_no)
```

## Parameters

module	Specified module address (1...15).	LONG
adc_no	Number of the ADC to be read (1...4 or 1...8).	LONG
ret_val	Measurement value in the F-ADC register (0...65535).	LONG

## Notes

The instructions **P2\_Read\_ADCF4**, **P2\_Read\_ADCF8**, **P2\_Read\_ADCF4\_Packed**, **P2\_Read\_ADCF8\_Packed** read conversion results very fast.

## See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#), [P2\\_ADCF\\_Mode](#), [P2\\_ADCF\\_Read\\_Limit](#), [P2\\_ADCF\\_Set\\_Limit](#), [P2\\_Read\\_ADCF32](#), [P2\\_Read\\_ADCF\\_SConv](#), [P2\\_Read\\_ADCF\\_SConv32](#), [P2\\_Read\\_ADCF4](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim value1 As Long          'Declaration

Event:
P2_Start_ConvF(1,1)         'Start AD conversion
P2_Wait_EOCF(1,1)           'Wait for end of conversion
value1 = P2_Read_ADCF(1,1) 'Read value from ADC
```

## P2\_Read\_ADCF

## P2\_Read\_ADCF24

**P2\_Read\_ADCF24** reads out the conversion result from an F-ADC of the specified module. The return value has a resolution of 24 bit.

### Syntax

```
#Include ADwinPro_All.Inc  
  
ret_val = P2_Read_ADCF24(module, adc_no)
```

### Parameters

module	Specified module address (1...15).	LONG
adc_no	Number of the ADC to be read (1...4 or 1...8).	LONG
ret_val	Measurement value in the F-ADC register (0...16777215 = $2^{24}-1$ ).	LONG

### Notes

The instructions **Read\_ADCF4\_24B**, **Read\_ADCF8\_24B** read conversion results very fast.

### See also

[P2\\_ADCF24](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#), [P2\\_ADCF\\_Mode](#), [P2\\_ADCF\\_Read\\_Limit](#), [P2\\_ADCF\\_Set\\_Limit](#), [P2\\_Read\\_ADCF\\_SConv24](#), [P2\\_Read\\_ADCF4\\_24B](#), [P2\\_Read\\_ADCF8\\_24B](#)

### Valid for

Aln-F-4/18 Rev. E, Aln-F-8/18 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
Dim value1 As Long           'Declaration  
  
Event:  
P2_Start_ConvF(1,1)          'Start AD conversion  
P2_Wait_EOCF(1,1)            'Wait for end of conversion  
value1 = P2_Read_ADCF24(1,1) 'Read 24 bit value from the ADC
```

**P2\_Read\_ADCF4** reads out the conversion results from the first 4 F-ADC of the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Read_ADCF4(module, array[], index)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[]</b>	Destination array, where conversion results are saved.	ARRAY LONG FLOAT
<b>index</b>	Element index in the destination array, where the first conversion result is saved.	LONG

## Notes

In any case the conversion results of the module's F-ADC 1...4 are read. The conversion results are saved subsequently in the destination array `array[]`, starting with element `index`.

The instruction is much faster than repeated use of **P2\_Read\_ADCF**.

## See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#), [P2\\_Read\\_ADCF](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#), [P2\\_Read\\_ADCF\\_SConv](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim value[4] As Long          'Array for conversion results

Init:
    P2_Start_ConvF(1,0Fh)      'Start AD conversion channels 1...4

Event:
    P2_Wait_EOCF(1,0Fh)        'Wait for end of conversion
    P2_Read_ADCF4(1,value,1)    'Read values of ADC 1...4
    P2_Start_ConvF(1,0Fh)      'Start new AD conversion
```

## P2\_Read\_ADCF4

## P2\_Read\_ADCF4\_24B

**P2\_Read\_ADCF4\_24B** reads out the conversion results from the first 4 F-ADC of the specified module. The return values have a resolution of 24 bits.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_Read_ADCF4_24B(module, array[], index)
```

### Parameters

module	Specified module address (1...15).	LONG
array[]	Destination array, where conversion results (24 bit) are saved.	ARRAY LONG FLOAT
index	Element index in the destination array, where the first conversion result is saved.	LONG

### Notes

In any case the conversion results of the module's F-ADC 1...4 are read. The conversion results are saved subsequently in the destination array `array[]`, starting with element `index`.

The instruction is much faster than repeated use of **P2\_Read\_ADCF24**.

### See also

[P2\\_ADCF24](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#), [P2\\_Read\\_ADCF](#), [P2\\_Read\\_ADCF\\_SConv24](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#), [P2\\_Read\\_ADCF8\\_24B](#)

### Valid for

Aln-F-4/18 Rev. E, Aln-F-8/18 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
Dim value[4] As Long           'Array for conversion results  
  
Init:  
    P2_Start_ConvF(1,0Fh)      'Start AD conversion channels 1...4  
  
Event:  
    P2_Wait_EOCF(1,0Fh)        'Wait for end of conversion  
    P2_Read_ADCF4_24B(1,value,1)'Read values of ADC 1...4  
    P2_Start_ConvF(1,0Fh)      'Start new AD conversion
```

**P2\_Read\_ADCF8** reads out the conversion results from all 8 F-ADCs of the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Read_ADCF8(module, array[], index)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[]</b>	Destination array, where conversion results are saved.	ARRAY LONG FLOAT
<b>index</b>	Element index in the destination array, where the first conversion result is saved.	LONG

## Notes

In any case the conversion results of the module's F-ADC 1...8 are read. The conversion results are saved subsequently in the destination array `array[]`, starting with element `index`.

The instruction is much faster than repeated use of **P2\_Read\_ADCF**.

## See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#), [P2\\_Read\\_ADCF4](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#), [P2\\_Read\\_ADCF\\_SConv](#)

## Valid for

Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim value[8] As Long          'Array for conversion results

Init:
    P2_Start_ConvF(1,0FFh)    'Start AD conversion channels 1...8

Event:
    P2_Wait_EOCF(1,0FFh)      'Wait for end of conversion
    P2_Read_ADCF8(1,value,1)  'Read values of ADC 1...8
    P2_Start_ConvF(1,0FFh)    'Start new AD conversion
```

## P2\_Read\_ADCF8

## P2\_Read\_ADCF8\_24B

**P2\_Read\_ADCF8\_24B** reads out the conversion results from all 8 F-ADC of the specified module. The return values have a resolution of 24 bits.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Read_ADCF8_24B(module, array[], index)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[]</b>	Destination array, where conversion results (24 bit) are saved.	ARRAY LONG FLOAT
<b>index</b>	Element index in the destination array, where the first conversion result is saved.	LONG

### Notes

In any case the conversion results of the module's F-ADC 1...8 are read. The conversion results are saved subsequently in the destination array `array[]`, starting with element `index`.

The instruction is much faster than repeated use of **P2\_Read\_ADCF24**.

### See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#), [P2\\_Read\\_ADCF\\_SConv24](#)

### Valid for

Aln-F-8/18 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim value[8] As Long           'Array for conversion results

Init:
    P2_Start_ConvF(1,0FFh)     'Start AD conversion channels 1...8

Event:
    P2_Wait_EOCF(1,0FFh)       'Wait for end of conversion
    P2_Read_ADCF8_24B(1,value,1)'Read values of ADC 1...8
    P2_Start_ConvF(1,0FFh)     'Start new AD conversion
```

**P2\_Read\_ADCF4\_Packed** reads out the conversion results from the first 4 F-ADC of the specified module.

Every 2 consecutive F-ADC results are returned in a single 32-bit value.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Read_ADCF4_Packed(module, array[], index)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[]</b>	Destination array, where conversion results are saved.	ARRAY LONG FLOAT
<b>index</b>	Element index in the destination array, where the first conversion result is saved.	LONG

## Notes

In any case the conversion results of the module's F-ADC 1...4 are read. The conversion result of an F-ADC with odd number is written into the lower word, of an F-ADC with even number into the higher word. The values are saved into the destination array `array[]` as follows:

Array element no.	Bit no.	
	31...16	15...0
<code>index</code>	F-ADC 2	F-ADC 1
<code>index+1</code>	F-ADC 4	F-ADC 3

## See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#), [P2\\_Read\\_ADCF\\_SConv](#), [P2\\_Read\\_ADCF4](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF8\\_Packed](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim value[4] As Long      'Array for conversion results

Init:
    P2_Start_ConvF(1,0Fh)  'Start AD conversion channels 1...4

Event:
    P2_Wait_EOCF(1,0Fh)    'Wait for end of conversion
    P2_Read_ADCF4_Packed(1,value,1) 'Read values of ADC 1...4
    P2_Start_ConvF(1,0Fh)  'Start new AD conversion
```

## P2\_Read\_ADCF4\_Packed

## P2\_Read\_ADCF8\_Packed

**P2\_Read\_ADCF8\_Packed** reads out the conversion results from all 8 F-ADC of the specified module.

Every 2 consecutive F-ADC results are returned in a single 32-bit value.

### Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_Read_ADCF8_Packed(module, array[], index)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[]</b>	Destination array, where conversion results are saved.	ARRAY LONG FLOAT
<b>index</b>	Element index in the destination array, where the first conversion result is saved.	LONG

### Notes

In any case the conversion results of the module's F-ADC 1...8 are read. The conversion result of an F-ADC with odd number is written into the lower word, of an F-ADC with even number into the higher word. The values are saved into the destination array `array[]` as follows:

Array element no.	Bit no.	
	31...16	15...0
<code>index</code>	F-ADC 2	F-ADC 1
<code>index+1</code>	F-ADC 4	F-ADC 3
<code>index+2</code>	F-ADC 6	F-ADC 5
<code>index+3</code>	F-ADC 8	F-ADC 7

### See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#), [P2\\_Read\\_ADCF\\_SConv](#), [P2\\_Read\\_ADCF4](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF4\\_Packed](#)

### Valid for

Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim value[8] As Long 'Array for conversion results

Init:
    P2_Start_ConvF(1,0Fh) 'Start AD conversion channels 1...8

Event:
    P2_Wait_EOCF(1,0Fh) 'Wait for end of conversion
    P2_Read_ADCF8_Packed(1,value,1) 'Read values of ADC 1...8
    P2_Start_ConvF(1,0Fh) 'Start new AD conversion
```



**P2\_Read\_ADCF32** reads the conversion result from 2 consecutive F-ADCs of the specified module and returns them in a single 32-bit value.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Read_ADCF32(module, adc_no)
```

## Parameters

module	Specified module address (1...15).	LONG										
adc_no	Number (1...2 or 1...4) of the pair of F-ADC to read:	LONG										
	<table><tr><td>adc_no</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>F-ADC-no.</td><td>1, 2</td><td>3, 4</td><td>5, 6</td><td>7, 8</td></tr></table>	adc_no	1	2	3	4	F-ADC-no.	1, 2	3, 4	5, 6	7, 8	
adc_no	1	2	3	4								
F-ADC-no.	1, 2	3, 4	5, 6	7, 8								
ret_val	The measurement values in the F-ADC registers (0...65535 each); one measurement value in the lower and one in the higher word.	LONG										

## Notes

The conversion result of the ADC with the number **adc\_no** is written into the lower word, the result of the ADC **adc\_no+1** into the higher word.

The number of the first F-ADC must be odd. Therefore it is for instance not possible to read out the conversion results of the F-ADCs 2 and 3 with one instruction.

## See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#), [P2\\_Read\\_ADCF](#), [P2\\_Read\\_ADCF\\_SConv](#), [P2\\_Read\\_ADCF\\_SConv32](#), [P2\\_Read\\_ADCF4](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#)

## Valid for

Aln-F-4/14 Rev. E, Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim value1 As Long 'Declaration

Event:
P2_Start_ConvF(1,3) 'Start AD conversion on ADC1 and ADC2
P2_Wait_EOCF(1,3) 'Wait for the end of conversions
value1 = P2_Read_ADCF32(1,1) 'Read values of ADC1 and ADC2
```

## P2\_Read\_ADCF32

## P2\_Read\_ADCF\_SConv

**P2\_Read\_ADCF\_SConv** reads the conversion result from an F-ADC of the specified module and immediately starts a new conversion.

### Syntax

```
#Include ADwinPro_All.Inc  
  
ret_val = P2_Read_ADCF_SConv(module, adc_no)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>adc_no</b>	Number of the ADC to be read (1...4 or 1...8).	LONG
<b>ret_val</b>	Measurement value in the F-ADC register (0...65535).	LONG

### See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#), [P2\\_Read\\_ADCF32](#), [P2\\_Read\\_ADCF\\_SConv32](#), [P2\\_Read\\_ADCF4](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#)

### Valid for

Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
Dim i As Long  
Dim Data_1[1000] As Long 'Declaration  
  
Init:  
  i=1  
  P2_Start_ConvF(1,1) 'Start A/D converter  
  
Event:  
  P2_Wait_EOCF(1,1)  
  Data_1[i] = P2_Read_ADCF_SConv(1,1) 'Read and start ADC  
  Inc(i) 'Increment index  
  If (i=1001) Then End 'End process after 1000 measurement  
  'values
```

**P2\_Read\_ADCF\_SConv24** reads the conversion result from an F-ADC of the specified module and immediately starts a new conversion.

The return value has a resolution of 24 bit.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Read_ADCF_SConv24(module, adc_no)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>adc_no</b>	Number of the ADC to be read (1...4 or 1...8).	LONG
<b>ret_val</b>	Measurement value in the F-ADC register (0...16777215 = $2^{24}-1$ ).	LONG

## See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Read\\_ADCF](#), [P2\\_Read\\_ADCF4](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#), [P2\\_Read\\_ADCF32](#), [P2\\_Read\\_ADCF\\_SConv32](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#)

## Valid for

Aln-F-4/18 Rev. E, Aln-F-8/18 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim i As Long
Dim Data_1[1000] As Long 'Declaration

Init:
    i=1
    P2_Start_ConvF(1,1) 'start A/D conversion

Event:
    P2_Wait_EOCF(1,1)
    Data_1[i] = P2_Read_ADCF_SConv24(1,1) 'Read out + start
                                           'AD converter 24 bit
    Inc(i) 'Increment index
    If (i=1001) Then End 'End process after 1000 measurement
                        ''values
```

## P2\_Read\_ADCF\_SConv24

## P2\_Read\_ADCF\_SConv32

**P2\_Read\_ADCF\_SConv32** reads the conversion results from 2 F-ADCs of the specified module and returns them in a 32-bit value.

Then a new conversion is started immediately.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Read_ADCF_SConv32(module, adc_no)
```

### Parameters

module	Specified module address (1...15).	LONG										
adc_no	Number of the first F-ADC to read: 1...2 or 1...4.	LONG										
<table><tr><td>adc_no</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>F-ADC-no.</td><td>1, 2</td><td>3, 4</td><td>5, 6</td><td>7, 8</td></tr></table>			adc_no	1	2	3	4	F-ADC-no.	1, 2	3, 4	5, 6	7, 8
adc_no	1	2	3	4								
F-ADC-no.	1, 2	3, 4	5, 6	7, 8								
ret_val	The return value (32-bit) contains the measurement data of 2 consecutive F-ADCs (16-bit each: 0...65535); one measurement value is in the lower word and one in the upper word.	LONG										

### See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Read\\_ADCF](#), [P2\\_Read\\_ADCF4](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#), [P2\\_Read\\_ADCF32](#), [P2\\_Read\\_ADCF\\_SConv](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#)

### Valid for

Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim value As Long 'Declaration

Init:
    P2_Start_ConvF(1,3) 'Start AD conversion

Event:
    P2_Wait_EOCF(1,3) 'Wait for end of conversion
    value = P2_Read_ADCF_SConv32(1,1) 'read values from ADC1 and
                                     'ADC2, start conversion of both ADCs
```

**P2\_Set\_Gain** sets the operating mode of a channel on the selected module and thus the gain and measurement range.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Set_Gain(module, channel, mode)
```

## Parameters

**module** Specified module address (1...15). LONG

**channel** Number of the ADC (1...4 or 1...8) where the gain is to be set. LONG

**mode** Operating mode (0...1) of the ADC: Sets the gain of the input signal. With gain, the measurement range of input signals changes proportionally. LONG

Operating mode <i>mode</i>	Gain $2^n$	Measurement range $\pm 10V / 2^n$
0	1	$\pm 10V$
1	2	$\pm 5V$
2	4	$\pm 2.5V$
3	8	$\pm 1.25V$

## See also

[P2\\_ADCF24](#), [P2\\_Read\\_ADCF](#), [P2\\_Start\\_ConvF](#), [P2\\_Wait\\_EOCF](#)

## Valid for

Aln-F-4/16 Rev. E, Aln-F-8/16 Rev. E

## Example

```
#Include ADwinPro_All.Inc
#Define ainadr 1           'Module address AIN module

Init:
  Rem Set voltage range of channel 4 to mode 1
  Rem Measuring range: +5V...-5V
  P2_Set_Gain(ainadr,4,1)

Event:
  Par_1 = P2_ADCF(1,4)     'Measure analog input 4
```

## P2\_Set\_Gain

## P2\_Start\_ConvF

**P2\_Start\_ConvF** starts the conversion on one or more F-ADCs of the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Start_ConvF(module, adc_no)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>adc_no</b>	Bit pattern that determines the ADCs whose conversion is to be started (see table).	LONG

Bit no.	31:8	7	6	5	4	3	2	1	0
Converter no.	–	8	7	6	5	4	3	2	1

### Notes

Determining the ADCs is made bit by bit, so that the conversion can be started from several converters at the same time. For instance, when starting the A/D converters 1 and 3 the bit pattern **101b** (decimal 5) has to be transferred.

With module Aln-F-x/14, **P2\_Start\_ConvF** is not required, since the ADCs run continuously with fixed conversion rate. Attention: If the instruction is used yet, the current measurement value is copied into the latch register; a following **P2\_Read\_ADCF** will read the copied value, even if this happens much later.

You can start a conversion with the instruction **P2\_Sync\_All** synchronously with other measurements, if you have released the module with **P2\_Sync\_Enable** for synchronization.

Several conversions can also be executed synchronously, if you have released the corresponding modules with **P2\_Sync\_Mode** for synchronization.

As soon as you start a conversion on the master module, you start simultaneously conversions on all channels of the slave modules. You will have the same effect with event-controlled modules, as soon as a signal is provided at the event-input.

### See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Read\\_ADCF](#), [P2\\_Read\\_ADCF4](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#), [P2\\_Wait\\_EOCF](#), [P2\\_Sync\\_All](#), [P2\\_Sync\\_Mode](#)

### Valid for

Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim value As Long 'Declaration

Event:
P2_Start_ConvF(1,1) 'Start AD conversion channel 1
P2_Wait_EOCF(1,1) 'Wait for end of conversion
value = P2_Read_ADCF(1,1) 'Read value from ADC
```

**P2\_Wait\_EOCF** waits until the end of conversion on all F-ADCs of the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Wait_EOCF(module, adc_no)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>adc_no</b>	Bit pattern that determines the ADCs, whose end of conversion shall be awaited (see table).	LONG

Bit no.	31:8	7	6	5	4	3	2	1	0
Converter no.	–	8	7	6	5	4	3	2	1

## Notes

Determining the ADCs is made bit by bit, so that the conversion can be started from several converters at the same time. For instance, when starting the A/D converters 1 and 3 the bit pattern **101b** (decimal 5) has to be transferred.

With module Aln-F-x/14, **P2\_Wait\_ConvF** is not required, since the ADCs run continuously with fixed conversion rate. The instruction has no effect except for loss of processor time.

## See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_Start\\_ConvF](#), [P2\\_Read\\_ADCF](#), [P2\\_Read\\_ADCF4](#), [P2\\_Read\\_ADCF8](#), [P2\\_Read\\_ADCF4\\_Packed](#), [P2\\_Read\\_ADCF8\\_Packed](#)

## Valid for

Aln-F-4/16 Rev. E, Aln-F-4/18 Rev. E, Aln-F-8/16 Rev. E, Aln-F-8/18 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim value As Long 'Declaration

Event:
P2_Start_ConvF(1,1) 'Start AD conversion
P2_Wait_EOCF(1,1) 'Wait for end of conversion
value = P2_Read_ADCF(1,1) 'Read value from ADC
```

## P2\_Wait\_EOCF

### 3.4 Pro II: Output Modules

This section describes instructions which apply to Pro II analog output modules:

- [P2\\_DAC](#) (page 117)
- [P2\\_DAC4](#) (page 118)
- [P2\\_DAC4\\_Packed](#) (page 119)
- [P2\\_DAC8](#) (page 121)
- [P2\\_DAC8\\_Packed](#) (page 122)
- [P2\\_Start\\_DAC](#) (page 124)
- [P2\\_Write\\_DAC](#) (page 125)
- [P2\\_Write\\_DAC4](#) (page 126)
- [P2\\_Write\\_DAC4\\_Packed](#) (page 127)
- [P2\\_Write\\_DAC8](#) (page 128)
- [P2\\_Write\\_DAC8\\_Packed](#) (page 129)
- [P2\\_Write\\_DAC32](#) (page 130)

In the [Instruction List sorted by Module Types](#) (annex A.2) you will find which of the functions corresponds to the *ADwin-Pro II* modules.

It is presumed that application examples use the module address 1 for D/A modules.





**P2\_DAC** outputs an (analog) voltage on the channel of the specified module, that corresponds to the indicated digital value.

## Syntax

```
#Include ADwinPro_All.Inc

P2_DAC(module, dac_no, value)
```

## Parameters

module	Specified module address (1...15).	LONG
dac_no	Number (1...4 or 1...8) of the output.	LONG
value	value to output (0...65535).	LONG

## Notes

We recommend to use the instructions **P2\_DAC4** or **P2\_DAC8** instead, since they can output more values than **P2\_DAC** in the same time.

**P2\_DAC** is characterized by a sequence of several commands:

<b>P2_Write_DAC</b>	→	<b>P2_Start_DAC</b>
Transfer digital value into DAC register		Start D/A conversion

## See also

[P2\\_DAC4](#), [P2\\_DAC4\\_Packed](#), [P2\\_DAC8](#), [P2\\_DAC8\\_Packed](#), [P2\\_Start\\_DAC](#), [P2\\_Write\\_DAC](#), [P2\\_Write\\_DAC4](#), [P2\\_Write\\_DAC4\\_Packed](#), [P2\\_Write\\_DAC8](#), [P2\\_Write\\_DAC8\\_Packed](#), [P2\\_Write\\_DAC32](#)

## Valid for

AOut-4/16 Rev. E, AOut-8/16 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

*REM Digital proportionl controller*

```
#Include ADwinPro_All.Inc
Dim sp, dev As Long
Dim g, actuate As Long
```

### Event:

```
sp = Par_1 'setpoint
g = Par_2 'Gain
dev = sp - P2_ADC(1,1) 'Calculate control deviation
actuate = dev * g 'Calculate actuating value
P2_DAC(1,1,actuate) 'Output of actuating value
```

## P2\_DAC

## P2\_DAC4

**P2\_DAC4** outputs 4 digital values from an array on the DAC 1...4 of the specified module as (analog) voltages.

### Syntax

```
#Include ADwinPro_All.Inc

P2_DAC4(module,array[ ],index)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[ ]</b>	Array with values (0...65535) to be output.	ARRAY
		LONG
		FLOAT
<b>index</b>	Index of the first array element to be output.	LONG

### Notes

**P2\_DAC4** is characterized by a sequence of several commands:

<b>P2_Write_DAC4</b>	→	<b>P2_Start_DAC</b>
Transfer digital value into DAC register		Start D/A conversion.

### See also

[P2\\_DAC](#), [P2\\_DAC4\\_Packed](#), [P2\\_DAC8](#), [P2\\_DAC8\\_Packed](#), [P2\\_Start\\_DAC](#), [P2\\_Write\\_DAC](#), [P2\\_Write\\_DAC4](#), [P2\\_Write\\_DAC4\\_Packed](#), [P2\\_Write\\_DAC8](#), [P2\\_Write\\_DAC8\\_Packed](#), [P2\\_Write\\_DAC32](#)

### Valid for

AOut-4/16 Rev. E, AOut-8/16 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

*REM Digital proportionl controller for 4 channels*

```
#Include ADwinPro_All.Inc
Dim sp, dev As Long
Dim i, g, actuate As Long
Dim array[4] As Long
```

#### Event:

```
sp = Par_1           'setpoint
g = Par_2           'Gain
P2_Read_ADCF4(1,array,1) 'read 4 input values
For i = 1 To 4
    dev = sp - array[i] 'Calculate control deviation
    array[i] = dev * g 'Calculate actuating value
Next i
P2_DAC4(2,array,1) 'output 4 actuating values
```

**P2\_DAC4\_Packed** outputs 4 packed digital values from an array on the DAC 1...4 of the specified module as (analog) voltages.

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_DAC4_Packed(module,array[],index)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[]</b>	Array with values (0...65535) to be output as packed data: Each 32 Bit array element holds 2 values of 16 Bit.	ARRAY LONG FLOAT
<b>index</b>	Index ( <sup>31</sup> ) of the first array element to be output.	LONG

## Notes

**P2\_DAC4\_Packed** is characterized by a sequence of two commands:

<b>P2_Write_DAC4_Packed</b>	→	<b>P2_Start_DAC</b>
Transfer 4 digital values into DAC registers.		Start D/A conversion.

Every 2 array elements of 32 Bit hold 4 digital values of 16 Bit in the following order:

Array element	array[n+1]		array[n]	
Bit no.	31:16	15:00	31:16	15:00
Digital value for	DAC4	DAC3	DAC2	DAC1

## See also

[P2\\_DAC](#), [P2\\_DAC4](#), [P2\\_DAC8](#), [P2\\_DAC8\\_Packed](#), [P2\\_Start\\_DAC](#), [P2\\_Write\\_DAC](#), [P2\\_Write\\_DAC4](#), [P2\\_Write\\_DAC4\\_Packed](#), [P2\\_Write\\_DAC8](#), [P2\\_Write\\_DAC8\\_Packed](#), [P2\\_Write\\_DAC32](#)

## Valid for

AOut-4/16 Rev. E, AOut-8/16 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## P2\_DAC4\_Packed

## Example

```
REM Digital proportionl controller for 4 channels
#include ADwinPro_All.inc
Dim sp, dev1, dev2 As Long
Dim i, g As Long
Dim array[2] As Long

Event:
  sp = Par_1          'setpoint
  g = Par_2          'gain
  P2_Read_ADCF4_Packed(1,array,1)'read 4 input values
  For i = 1 To 2
    REM Calculare control deviations
    dev1 = sp - (array[i] And 0FFh)
    dev2 = sp - (Shift_Right(array[i],16) And 0FFh)
    REM Calculate actuating values and store
    array[i] = Shift_Left(dev2*g, 16) + dev1*g
  Next i
  P2_DAC4_Packed(2,array,1)'output 4 actuating values
```

**P2\_DAC8** outputs 8 digital values from an array on the DAC 1...8 of the specified module as (analog) voltages.

## Syntax

```
#Include ADwinPro_All.Inc

P2_DAC8(module,array[],index)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[]</b>	Array with values (0...65535) to be output.	ARRAY
		LONG
		FLOAT
<b>index</b>	Index ( <sup>31</sup> ) of the first array element to be written.	LONG

## Notes

**P2\_DAC8** is characterized by a sequence of several commands:

<b>P2_Write_DAC8</b>	→	<b>P2_Start_DAC</b>
transfer 8 digital values into the DAC register.		Start D/A conversion.

## See also

[P2\\_DAC](#), [P2\\_DAC4](#), [P2\\_DAC4\\_Packed](#), [P2\\_DAC8\\_Packed](#), [P2\\_Start\\_DAC](#), [P2\\_Write\\_DAC](#), [P2\\_Write\\_DAC4](#), [P2\\_Write\\_DAC4\\_Packed](#), [P2\\_Write\\_DAC8](#), [P2\\_Write\\_DAC8\\_Packed](#), [P2\\_Write\\_DAC32](#)

## Valid for

AOut-8/16 Rev. E

## Example

*REM Digital proportionl controller for 4 channels*

```
#Include ADwinPro_All.Inc
Dim sp, dev As Long
Dim i, g, actuate As Long
Dim array[8] As Long
```

### Event:

```
sp = Par_1           'setpoint
g = Par_2           'Gain
P2_Read_ADCF8(1,array,1) 'read 8 input values
For i = 1 To 8
    dev = sp - array[i] 'Calculate control deviation
    array[i] = dev * g 'Calculate actuating value
Next i
P2_DAC8(2,array,1) 'output 8 actuating values
```

## P2\_DAC8

## P2\_DAC8\_Packed

**P2\_DAC8\_Packed** outputs 8 packed digital values from an array on the DAC 1...8 of the specified module as (analog) voltages.

### Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_DAC8_Packed(module,array[ ],index)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[ ]</b>	Array with values (0...65535) to be output as packed data: Each 32 bit array element holds 2 values of 16 bit.	ARRAY LONG FLOAT
<b>index</b>	Index ( <sup>31</sup> ) of the first array element to be written.	LONG

### Notes

**P2\_DAC8\_Packed** is characterized by a sequence of two commands:

**P2\_Write\_DAC8** → **P2\_Start\_DAC**

Transfer digital value into DAC register

Start D/A conversion.

Every 4 array elements of 32 Bit hold 8 digital values of 16 Bit in the following order:

Array element	array[n+3]		array[n+2]		array[n+1]		array[n]	
Bit no.	31:16	15:00	31:16	15:00	31:16	15:00	31:16	15:00
Digital value for	DAC8	DAC7	DAC6	DAC5	DAC4	DAC3	DAC2	DAC1

### See also

[P2\\_DAC](#), [P2\\_DAC4](#), [P2\\_DAC4\\_Packed](#), [P2\\_DAC8](#), [P2\\_Start\\_DAC](#), [P2\\_Write\\_DAC](#), [P2\\_Write\\_DAC4](#), [P2\\_Write\\_DAC4\\_Packed](#), [P2\\_Write\\_DAC8](#), [P2\\_Write\\_DAC8\\_Packed](#), [P2\\_Write\\_DAC32](#)

### Valid for

AOut-8/16 Rev. E

### Example

```
REM Digital proportionl controller for 8 channels
#include ADwinPro_All.Inc
Dim sp, dev1, dev2 As Long
Dim i, g As Long
Dim array[4] As Long

Event:
  sp = Par_1          'setpoint
  g = Par_2          'Gain
  P2_Read_ADCF8_Packed(1,array,1)'read 8 input values
  For i = 1 To 4
    REM Calculate control deviations
    dev1 = sp - (array[i] And 0FFh)
    dev2 = sp - (Shift_Right(array[i],16) And 0FFh)
    REM Calculate setpoints and store
    array[i] = Shift_Left(dev2*g, 16) + dev1*g
  Next i
  P2_DAC8_Packed(2,array,1) 'output 8 setpoints
```

## P2\_Start\_DAC

**P2\_Start\_DAC** starts the conversion or output of all DAC on the specified module

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_Start_DAC(module)
```

### Parameters

**module** Specified module address (1...15).

LONG

### Notes

The conversion can be started synchronously to other modules. If so, use **P2\_Sync\_All**.

### See also

[P2\\_DAC](#), [P2\\_DAC4](#), [P2\\_DAC4\\_Packed](#), [P2\\_DAC8](#), [P2\\_DAC8\\_Packed](#), [P2\\_Write\\_DAC](#), [P2\\_Write\\_DAC4](#), [P2\\_Write\\_DAC4\\_Packed](#), [P2\\_Write\\_DAC8](#), [P2\\_Write\\_DAC8\\_Packed](#), [P2\\_Write\\_DAC32](#)

### Valid for

AOut-4/16 Rev. E, AOut-8/16 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

*REM Simultaneous output of two different signals  
REM on the outputs 1 and 2 of a D/A module.*

```
#Include ADwinPro_All.Inc
```

```
Dim i As Long
```

```
Init:
```

```
    i = 0
```

```
Event:
```

```
    P2_Write_DAC(1,1,i)      'Set output register DAC1
```

```
    P2_Write_DAC(1,2,65535-i)'Set output register DAC2
```

```
    P2_Start_DAC(1)         'Start output on all DAC
```

```
    Inc(i)
```

```
    If (i=65535) Then i=0
```



**P2\_Write\_DAC** writes a digital value into the output register of a DAC on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Write_DAC(module,dac_no,value)
```

## Parameters

module	Specified module address (1...15).	LONG
dac_no	Number (1...4 or 1...8) of the output.	LONG
value	value to output (0...65535).	LONG

## Notes

**P2\_Start\_DAC** starts the conversion of the digital value into an output voltage.

We recommend to use the instructions **P2\_Write\_DAC4** or **P2\_Write\_DAC8** instead, since they can output more values than **P2\_Write\_DAC** in the same time.

## See also

[P2\\_DAC](#), [P2\\_DAC4](#), [P2\\_DAC4\\_Packed](#), [P2\\_DAC8](#), [P2\\_DAC8\\_Packed](#), [P2\\_Start\\_DAC](#), [P2\\_Write\\_DAC4](#), [P2\\_Write\\_DAC4\\_Packed](#), [P2\\_Write\\_DAC8](#), [P2\\_Write\\_DAC8\\_Packed](#), [P2\\_Write\\_DAC32](#)

## Valid for

AOut-4/16 Rev. E, AOut-8/16 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
REM Simultaneous output of four different signals
REM on the output channels 1, 2, 3 and 4 of a D/A module
REM The signals are filed in four DATA arrays and
REM can be transferred from the PC before program start
```

```
#Include ADwinPro_All.Inc
Dim i As Long
Dim Data_1[1000], Data_2[1000], Data_3[1000] As Long
Dim Data_4[1000] As Long
```

### Init:

```
i = 1
```

### Event:

```
P2_Write_DAC(1,1,Data_1[i])'Set output register DAC1
P2_Write_DAC(1,2,Data_2[i])'Set output register DAC2
P2_Write_DAC(1,3,Data_3[i])'Set output register DAC3
P2_Write_DAC(1,4,Data_4[i])'Set output register DAC4
P2_Start_DAC(1)           'Start output on all DAC
Inc(i)
If (i>1000) Then i = 1
```

## P2\_Write\_DAC

## P2\_Write\_DAC4

**P2\_Write\_DAC4** writes 4 digital values from an array into the output registers of the DAC 1...4 of the specified module.

**P2\_Start\_DAC** starts the conversion of the digital values into the output voltages.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Write_DAC4(module,array[ ],index)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[ ]</b>	Array with values (0...65535) to be output.	ARRAY
		LONG
		FLOAT
<b>index</b>	Index of the first array element to be output.	LONG

### See also

[P2\\_DAC](#), [P2\\_DAC4](#), [P2\\_DAC4\\_Packed](#), [P2\\_DAC8](#), [P2\\_DAC8\\_Packed](#), [P2\\_Start\\_DAC](#), [P2\\_Write\\_DAC](#), [P2\\_Write\\_DAC4\\_Packed](#), [P2\\_Write\\_DAC8](#), [P2\\_Write\\_DAC8\\_Packed](#), [P2\\_Write\\_DAC32](#)

### Valid for

AOut-4/16 Rev. E, AOut-8/16 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

### Example

*REM Simultaneous output of four different signals  
REM on the output channels 1, 2, 3 and 4 of a D/A module  
REM The signals are stored sequentially in the array Data\_1  
REM and may be transferred before program start from the PC*

```
#Include ADwinPro_All.Inc
Dim i As Long
Dim Data_1[4000] As Long
```

```
Init:
    i = 1
```

```
Event:
    REM Set output registers DAC1...DAC4
    P2_Write_DAC4(1,Data_1,(i-1)*4+i)
    P2_Start_DAC(1) 'Start output on all DAC
    Inc(i)
    If (i>1000) Then i = 1
```

**P2\_Write\_DAC4\_Packed** writes 4 packed digital values from an array into the output registers of the DAC 1...4 of the specified module.

**P2\_Start\_DAC** starts the conversion of the digital values into the output voltages.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Write_DAC4_Packed(module,array[],index)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[]</b>	Array with values (0...65535) to be output as packed data: Each 32 bit array element holds 2 values of 16 bit.	ARRAY LONG FLOAT
<b>index</b>	Index of the first array element to be output.	LONG

## Notes

Every 2 array elements of 32 Bit hold 4 digital values of 16 Bit in the following order:

Array element	array[n+1]		array[n]	
Bit no.	31:16	15:00	31:16	15:00
Digital value for	DAC4	DAC3	DAC2	DAC1

## See also

[P2\\_DAC](#), [P2\\_DAC4](#), [P2\\_DAC4\\_Packed](#), [P2\\_DAC8](#), [P2\\_DAC8\\_Packed](#), [P2\\_Start\\_DAC](#), [P2\\_Write\\_DAC](#), [P2\\_Write\\_DAC4](#), [P2\\_Write\\_DAC8](#), [P2\\_Write\\_DAC8\\_Packed](#), [P2\\_Write\\_DAC32](#)

## Valid for

AOut-4/16 Rev. E, AOut-8/16 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E

## Example

```
REM Simultaneous output of four different signals
REM on the output channels 1, 2, 3 and 4 of a D/A module
REM The signals are stored sequentially in the array Data_1
REM packed and can be transferred from the PC before program start
```

```
#Include ADwinPro_All.Inc
Dim i As Long
Dim Data_1[4000] As Long

Init:
    i = 1

Event:
REM Set output registers DAC1...DAC4
P2_Write_DAC4_Packed(1,Data_1,(i-1)*2+i)
P2_Start_DAC(1)          'Start output on all DAC
Inc(i)
If (i>1000) Then i = 1
```

## P2\_Write\_DAC4\_Packed

## P2\_Write\_DAC8

**P2\_Write\_DAC8** writes 8 digital values from an array into the output registers of the DAC 1...8 of the specified module.

**P2\_Start\_DAC** starts the conversion of the digital values into the output voltages.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Write_DAC8(module,array[ ],index)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[ ]</b>	Array with values (0...65535) to be output.	ARRAY
		LONG
		FLOAT
<b>index</b>	Index of the first array element to be output.	LONG

### See also

[P2\\_DAC](#), [P2\\_DAC4](#), [P2\\_DAC4\\_Packed](#), [P2\\_DAC8](#), [P2\\_DAC8\\_Packed](#), [P2\\_Start\\_DAC](#), [P2\\_Write\\_DAC](#), [P2\\_Write\\_DAC4](#), [P2\\_Write\\_DAC4\\_Packed](#), [P2\\_Write\\_DAC8\\_Packed](#), [P2\\_Write\\_DAC32](#)

### Valid for

AOut-8/16 Rev. E

### Example

#### Example

*REM Simultaneous output of four different signals  
REM on the outputs 1...8 of a D/A module.  
REM The signals are sequentially stored into a DATA array  
REM and may be transferred before program start from the PC.*

```
#Include ADwinPro_All.Inc
Dim i As Long
Dim Data_1[8000] As Long

Init:
    i = 1

Event:
    REM Set output registers DAC1...DAC8
    P2_Write_DAC8(1,Data_1,(i-1)*8+i)
    P2_Start_DAC(1) 'Start output on all DAC
    Inc(i)
    If (i>1000) Then i = 1
```

**P2\_Write\_DAC8\_Packed** writes 8 packed digital values from an array into the output registers of the DAC 1...8 of the specified module.

**P2\_Start\_DAC** starts the conversion of the digital values into the output voltages.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Write_DAC8_Packed(module,array[ ],index)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[ ]</b>	Array with values (0...65535) to be output as packed data: Each 32 bit array element holds 2 values of 16 bit.	ARRAY LONG FLOAT
<b>index</b>	Index of the first array element to be output.	LONG

## Notes

Every 4 array elements of 32 Bit hold 8 digital values of 16 Bit in the following order:

Array element	array[n+3]		array[n+2]		array[n+1]		array[n]	
Bit no.	31:16	15:00	31:16	15:00	31:16	15:00	31:16	15:00
Digital value for	DAC8	DAC7	DAC6	DAC5	DAC4	DAC3	DAC2	DAC1

## See also

[P2\\_DAC](#), [P2\\_DAC4](#), [P2\\_DAC4\\_Packed](#), [P2\\_DAC8](#), [P2\\_DAC8\\_Packed](#), [P2\\_Start\\_DAC](#), [P2\\_Write\\_DAC](#), [P2\\_Write\\_DAC4](#), [P2\\_Write\\_DAC4\\_Packed](#), [P2\\_Write\\_DAC8](#), [P2\\_Write\\_DAC32](#)

## Valid for

AOut-8/16 Rev. E

## Example

### Example

*REM Simultaneous output of four different signals*  
*REM on the outputs 1...8 of a D/A module.*  
*REM The signals are sequentially stored into a DATA array*  
*REM packed and can be transferred from the PC before program start*  
*REM übergeben werden.*

```
#Include ADwinPro_All.Inc
Dim i As Long
Dim Data_1[8000] As Long
```

### Init:

```
i = 1
```

### Event:

```
REM Set output registers DAC1...DAC8
P2_Write_DAC8_Packed(1,Data_1,(i-1)*4+i)
P2_Start_DAC(1) 'Start output on all DAC
Inc(i)
If (i>1000) Then i = 1
```

## P2\_Write\_DAC8\_Packed

## P2\_Write\_DAC32

**P2\_Write\_DAC32** copies two 16 Bit values from a 32 Bit value into the output registers of a DAC pair of the specified module.

The conversion into an output voltage is done using **Start\_DAC**.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Write_DAC32(module,dac_no,value32)
```

### Parameter

<b>module</b>	Specified module address (1...15).	LONG
<b>dac_no</b>	Selection of DAC pair: 0: DAC 1 and 2 1: DAC 3 and 4 2: DAC 5 and 6 3: DAC 7 and 8	LONG
<b>value32</b>	Auszugebender Wert (0h...0FFFFFFFh).	LONG

### See also

The lower word (bits 0...15) of the digital **value32** is written into the DAC with odd number, the upper word (bits 16...31) into the DAC with even number.

### See also

[P2\\_DAC](#), [P2\\_DAC4](#), [P2\\_DAC4\\_Packed](#), [P2\\_DAC8](#), [P2\\_DAC8\\_Packed](#), [P2\\_Start\\_DAC](#), [P2\\_Write\\_DAC](#), [P2\\_Write\\_DAC4](#), [P2\\_Write\\_DAC4\\_Packed](#), [P2\\_Write\\_DAC8](#), [P2\\_Write\\_DAC8\\_Packed](#)

### Valid for

AOut-4/16 Rev. E, AOut-8/16 Rev. E, MIO-4 Rev. E, MIO-4-ET1 Rev. E, MIO-4-ET1 Rev. E

### Example

```
REM Simultaneous output of two different signals
REM on the outputs 3 and 4 of a D/A module.
REM The signals are filed in two DATA arrays and
REM can be transferred from the PC before program start
```

```
#Include ADwinPro_All.Inc
Dim i As Long 'Declaration
Dim Data_1[1000], Data_2[1000] As Long
Dim array[1000] As Long

Init:
  For i = 1 To 1000
    array[i] = Shift_Left(Data_2[i],16) + Data_1[i]
  Next i
  i = 1

Event:
  P2_Write_DAC32(1,2,array[i])'Set output register DAC 3+4
  P2_Start_DAC(1) 'Start output on all DAC
  Inc(i)
  If (i>1000) Then i=1
```

### 3.5 Pro II: Digital I/O Modules

This section describes instructions which apply to Pro II digital I/O modules:

- [P2\\_Dig\\_FIFO\\_Mode](#) (page 132)
- [P2\\_Dig\\_Latch](#) (page 133)
- [P2\\_Dig\\_Read\\_Latch](#) (page 134)
- [P2\\_Dig\\_Write\\_Latch](#) (page 135)
- [P2\\_Digin\\_Edge](#) (page 136)
- [P2\\_Digin\\_FIFO\\_Clear](#) (page 137)
- [P2\\_Digin\\_FIFO\\_Enable](#) (page 138)
- [P2\\_Digin\\_FIFO\\_Full](#) (page 140)
- [P2\\_Digin\\_FIFO\\_Read](#) (page 141)
- [P2\\_Digin\\_Fifo\\_Read\\_Fast](#) (page 143)
- [P2\\_Digin\\_FIFO\\_Read\\_Timer](#) (page 145)
- [P2\\_Digin\\_Long](#) (page 146)
- [P2\\_Digout](#) (page 147)
- [P2\\_Digout\\_Bits](#) (page 148)
- [P2\\_Digout\\_FIFO\\_Clear](#) (page 149)
- [P2\\_Digout\\_FIFO\\_Empty](#) (page 150)
- [P2\\_Digout\\_FIFO\\_Enable](#) (page 151)
- [P2\\_Digout\\_FIFO\\_Read\\_Timer](#) (page 152)
- [P2\\_Digout\\_FIFO\\_Start](#) (page 153)
- [P2\\_Digout\\_FIFO\\_Write](#) (page 154)
- [P2\\_Digout\\_Long](#) (page 156)
- [P2\\_Digout\\_Reset](#) (page 157)
- [P2\\_Digout\\_Set](#) (page 158)
- [P2\\_Digprog](#) (page 159)
- [P2\\_Get\\_Digout\\_Long](#) (page 160)

In the [Instruction List sorted by Module Types](#) (annex A.2) you will find which of the functions corresponds to the *ADwin-Pro II* modules.

It is presumed that application examples use the module address 1.



## P2\_Dig\_FIFO\_Mode

**P2\_Dig\_FIFO\_Mode** sets the FIFO operation mode on the specified module, input with edge detection or edge output.

### Syntax

```
#Include ADwinPro_All.inc

P2_Dig_FIFO_Mode(module, mode)
```

### Parameters

module	Specified module address (1...15).	LONG
mode	FIFO operation mode: 0: Input FIFO with edge detection. Default. 1: Output FIFO with edge output, time values with absolute reference. 3: Output FIFO with edge output, time values with relative reference.	LONG

### Notes

The output FIFO is available since revision DIO-32-TiCo Rev. E 03.

Time stamps of an output FIFO set the time when an edge is output. The time stamp value can be defined as absolute oder relative reference:

- Absolute value: The time stamp refers to the starting time 0 of the module counter (**P2\_Digout\_FIFO\_Start**).  
Using this mode, the current counter value can be read with **P2\_Digout\_FIFO\_Read\_Timer**.
- Relative value: The time stamp is counted relative to the previous time stamp.

### See also

[P2\\_Digin\\_FIFO\\_Enable](#), [P2\\_Digout\\_FIFO\\_Read\\_Timer](#), [P2\\_Digout\\_FIFO\\_Start](#), [P2\\_Digout\\_FIFO\\_Write](#)

### Valid for

DIO-32-TiCo Rev. E

### Example

```
#Include ADwinPro_All.inc
#Define module 2
Dim value[4] As Long

Init:
    Processdelay = 6000          '6000 x 3.3 ns = 20µs
    value[1] = 01b               'output value n
    value[2] = 5000              ' with output time 50 µs (relative)
    value[3] = 10b               'output value n+1
    value[4] = 7000              ' with output time 70 µs (relative)
    P2_Digprog(module,0Fh)      'set all channels as output
    P2_Dig_FIFO_Mode(module,3)  'Set FIFO as relative output
    P2_Digout_FIFO_Clear(module) 'clear FIFO
    P2_Digout_FIFO_Enable(module,11b) 'Enable output channels 0+1
    Rem write 2 value pairs into output FIFO and start output
    P2_Digout_FIFO_Write(module,2,value,1)
    P2_Digout_FIFO_Start(Shift_Left(1,module-1))

Event:
    Rem write new value pairs into FIFO, if possible
    If (P2_Digout_FIFO_Empty(module) > 2) Then
        P2_Digout_FIFO_Write(module,2,value,1)
    EndIf
```



**P2\_Dig\_Latch** transfers digital information from the inputs to the input latches and from the output latches to the outputs on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Dig_Latch(module)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
---------------	------------------------------------	------

## Notes

With digital inputs the instructions reads the input signals into the input latches. With digital outputs the instruction passes the values of the output latches to the outputs.

If the module is released for synchronization by **P2\_Sync\_Enable**, **P2\_Sync\_All** has the same functions as **P2\_Dig\_Latch**.

## See also

[P2\\_Dig\\_Read\\_Latch](#), [P2\\_Dig\\_Write\\_Latch](#), [P2\\_Digprog](#), [P2\\_Digin\\_Long](#), [P2\\_Digout\\_Bits](#), [P2\\_Digout](#), [P2\\_Digout\\_Long](#), [P2\\_Get\\_Digout\\_Long](#), [P2\\_Sync\\_All](#), [P2\\_Sync\\_Mode](#)

## Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, OPT-16 Rev. E, REL-16 Rev. E, TRA-16 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Init:
    REM Set channels 0...15 as outputs, 16...31 as inputs
    P2_Digprog(1,0011h)
    P2_Dig_Write_Latch(1,0) 'Set all output bits to 0

Event:
    P2_Dig_Latch(1)          'latch inputs, output content of
                             'output latches

    Rem further program
    Par_1 = P2_Dig_Read_Latch(1) 'read input bits and ...
    P2_Dig_Write_Latch(1,Par_1) 'output in next event cycle
```

## P2\_Dig\_Latch

## P2\_Dig\_Read\_Latch

**P2\_Dig\_Read\_Latch** returns the bits from the latch register for the digital inputs of the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Dig_Read_Latch(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Bit pattern. Each bit corresponds to a digital input (see table).	LONG

Bit no.	31	30	29	...	2	1	0
Input	31	30	29	...	2	1	0

### Notes

It is recommended to first program the specified channels as inputs using **P2\_Digprog**.

The current status of the digital inputs can be transferred to the latch register with the following instructions:

- **P2\_Dig\_Latch**
- **P2\_Sync\_All** (when activated)

### See also

[P2\\_Dig\\_Latch](#), [P2\\_Dig\\_Write\\_Latch](#), [P2\\_Digprog](#), [P2\\_Digin\\_Long](#), [P2\\_Sync\\_All](#), [P2\\_Sync\\_Mode](#)

### Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, OPT-16 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim value As Long
```

#### Init:

```
REM Set DIO15:00 of modules 1+2 as inputs
P2_Digprog(1,0000b)
P2_Digprog(2,0000b)
```

#### Event:

```
REM synchronously transfer the logic levels at the digital
REM inputs of both modules to the latch register
P2_Sync_All(11b)
Par_1 = P2_Dig_Read_Latch(1)'Read latch register of module 1
Par_2 = P2_Dig_Read_Latch(2)'Read latch register of module 2
```

**P2\_Dig\_Write\_Latch** writes a 32 bit value into the latch register for the digital outputs on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Dig_Write_Latch(module,pattern)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern. Each bit corresponds to a digital output (see table).	LONG

Bit no.	31	30	29	...	2	1	0
Input	31	30	29	...	2	1	0

## Notes

The specified channels must first be programmed as outputs using **P2\_Digprog**.

You may set the value of the latch register for digital outputs with **P2\_Digout**.

With module version TRA-16-G Rev. E, high level switches to ground, not to  $V_{CC}$ .

## See also

[P2\\_Dig\\_Latch](#), [P2\\_Digout](#), [P2\\_Digprog](#), [P2\\_Get\\_Digout\\_Long](#)

## Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, REL-16 Rev. E, TRA-16 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Init:
  P2_Digprog(1,1111b)      'Set DIO31:00 as outputs

Event:
  P2_Dig_Latch(1)          'Output values from the latch
                           'registers
  P2_Dig_Write_Latch(1,Par_1)'write long word into the output
                           'latch
```

## P2\_Dig\_Write\_Latch

## P2\_Digin\_Edge

**P2\_Digin\_Edge** returns whether a positive or negative edge has occurred on digital inputs of the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Digin_Edge(module, edge)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>edge</b>	Kind of detected edge: 1: Detect positive edge. 0: Detect negative edge.	LONG
<b>ret_val</b>	Bit pattern where each bits represent an edge occurred at an input. The mapping of bits to inputs is shown below. Bit = 1: An edge has occurred. Bit = 0: No edge occurred.	LONG

Bit no.	31	30	29	...	2	1	0
Input	31	30	29	...	2	1	0

### Notes

A set bit in **ret\_val** means, that a selected edge has been occurred at least once at the digital input since the previous query. Bit for output channels always return zero.

A query with **P2\_Digin\_Edge** resets all bits to zero.

### See also

[P2\\_Digin\\_FIFO\\_Clear](#), [P2\\_Digin\\_FIFO\\_Enable](#), [P2\\_Digin\\_FIFO\\_Full](#), [P2\\_Digin\\_FIFO\\_Read](#), [P2\\_Digin\\_FIFO\\_Read\\_Timer](#)

### Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, OPT-16 Rev. E

### Example

```
#Include ADwinPro_All.Inc

Init:
    P2_Digprog(1,1100b)          'channels 0:15 as inputs

Event:
    REM check positive and negative edges, mask out outputs
    Par_1 = P2_Digin_Edge(1,1) And 0Fh
    Par_2 = P2_Digin_Edge(1,0) And 0Fh

    REM output edge changes
    If (Par_1 + Par_2 > 0) Then
        P2_Digout_Bits(1,Shift_Left(Par_1,16),Shift_Left(Par_2,16))
    EndIf
```

**P2\_Digin\_FIFO\_Clear** clears the FIFO of the edge detection unit on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Digin_FIFO_Clear (module)
```

## Parameters

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

## Notes

- / -

## See also

[P2\\_Digin\\_FIFO\\_Enable](#), [P2\\_Digin\\_FIFO\\_Full](#), [P2\\_Digin\\_FIFO\\_Read](#),  
[P2\\_Digin\\_FIFO\\_Read\\_Timer](#), [P2\\_Digin\\_Edge](#)

## Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, OPT-16 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Dim Data_1[10000], Data_2[10000] As Long
Dim num, index As Long

Init:
P2_Digprog(1,1100b)      'channels 0:15 as inputs
P2_Digin_FIFO_Enable(1,0)'output edge edge detection off
P2_Digin_FIFO_Clear(1)   'clear FIFO
P2_Digin_FIFO_Enable(1,10011b)'edge detection channels 1,2,5
index = 1

Event:
num = P2_Digin_FIFO_Full(1)'number of value pairs
If (num>50) Then
    If (index+num>10000) Then index = 1
    Rem read value pairs
    P2_Digin_FIFO_Read(1, num, Data_1, Data_2, index)
    index=index + num
EndIf
```

## P2\_Digin\_FIFO\_Clear

## P2\_Digin\_FIFO\_Enable

**P2\_Digin\_FIFO\_Enable** determines, which input channels of the specified module the edge detection unit will monitor.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_Digin_FIFO_Enable(module, channels)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>channels</b>	Bit pattern which determines the input channels to be monitored.	LONG

Bit no.	31	30	29	...	2	1	0
Input	31	30	29	...	2	1	0

### Notes

Only input channels can be monitored. The channels are programmed as inputs or outputs with **P2\_Digprog**.

The FIFO should be set to edge detection mode with **P2\_Dig\_FIFO\_Mode**.

The edge detection unit checks each 10ns, if an edge has occurred at the selected input channels or if a level has been changed. If an edge has occurred, a pair of values is copied into an internal FIFO array:

- Value 1 contains the level status of all channels as bit pattern.
- Value 2 contains a time stamp, which is the current value of a 100MHz timer.

The FIFO array may contain 511 value pairs (level status and time stamp) in maximum. If and as long as the FIFO array is filled completely, any additional value pair cannot be saved and will thus be lost.

### See also

[P2\\_Dig\\_FIFO\\_Mode](#), [P2\\_Digin\\_FIFO\\_Clear](#), [P2\\_Digin\\_FIFO\\_Full](#), [P2\\_Digin\\_FIFO\\_Read](#), [P2\\_Digin\\_FIFO\\_Read\\_Timer](#), [P2\\_Digin\\_Edge](#), [P2\\_Digprog](#)

### Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, OPT-16 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Dim Data_1[10000], Data_2[10000] As Long
Dim num, index As Long

Init:
  P2_Digprog(1,1100b)      'channels 0:15 as inputs
  P2_Digin_FIFO_Enable(1,0)'output edge edge detection off
  P2_Digin_FIFO_Clear(1)   'clear FIFO
  P2_Digin_FIFO_Enable(1,10011b)'edge detection channels 1,2,5
  index = 1

Event:
  num = P2_Digin_FIFO_Full(1)'number of value pairs
  If (num>50) Then
    Rem read value pairs
    P2_Digin_FIFO_Read(1, num, Data_1, Data_2, index)
    index=index + num
    If (index>10000) Then index = 1
  EndIf
```

## P2\_Digin\_FIFO\_Full

**P2\_Digin\_FIFO\_Full** returns the number of saved value pairs in the FIFO of the edge detection unit.

### Syntax

```
#Include ADwinPro_All.Inc  
  
ret_value = P2_Digin_FIFO_Full(module)
```

### Parameters

module	Specified module address (1...15).	LONG
ret_value	Number (0...511) of saved value pairs in the FIFO.	LONG

### Notes

The FIFO array may contain 511 value pairs (level status and time stamp) in maximum. If and as long as the FIFO array is filled completely, any additional value pair cannot be saved and will thus be lost.

### See also

[P2\\_Digin\\_FIFO\\_Clear](#), [P2\\_Digin\\_FIFO\\_Enable](#), [P2\\_Digin\\_FIFO\\_Read](#), [P2\\_Digin\\_FIFO\\_Read\\_Timer](#), [P2\\_Digin\\_Edge](#)

### Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, OPT-16 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
  
Dim Data_1[10000], Data_2[10000] As Long  
Dim num, index As Long  
  
Init:  
P2_Digprog(1,1100b)      'channels 0:15 as inputs  
P2_Digin_FIFO_Enable(1,0)'output edge edge detection off  
P2_Digin_FIFO_Clear(1)   'clear FIFO  
P2_Digin_FIFO_Enable(1,10011b)'edge detection channels 1,2,5  
index = 1  
  
Event:  
num = P2_Digin_FIFO_Full(1)'number of value pairs  
If (num>50) Then  
    Rem read value pairs  
    P2_Digin_FIFO_Read(1, num, Data_1, Data_2, index)  
    index=index + num  
    If (index>10000) Then index = 1  
EndIf
```



**P2\_Digin\_FIFO\_Read** reads the value pairs from the FIFO of the edge detection unit and writes them into 2 arrays.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Digin_FIFO_Read(module, count, value[],
    timestamp[], start_index)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Number (1...511) of value pairs to be read.	LONG
<b>value[ ]</b>	Array where the level status bit patterns are written. Each level status bit corresponds to a digital input (see table below).	LONG ARRAY
<b>timestamp[ ]</b>	Array where time stamps are written.	LONG ARRAY
<b>start_index</b>	Start index for both arrays <b>value[ ]</b> and <b>timestamp[ ]</b> , where the first value are written.	LONG

Bit no.	31	30	29	...	2	1	0
Input	31	30	29	...	2	1	0

## Notes

The function corresponds to the faster version **P2\_Digin\_Fifo\_Read\_Fast** but returns the data more comfortably in 2 arrays.

No more value pairs may be read than are saved in the FIFO. Thus, before reading there must be a check with **P2\_Digin\_FIFO\_Full**, how much value pairs are saved in the FIFO.

The arrays must be dimensioned with sufficient size, so that all value pairs may be written into the arrays.

The time difference between 2 level status patterns is the difference of the appropriate time stamps, measured in units of 10ns:

$$\Delta t = (\text{stamp}_1 - \text{stamp}_2) \cdot 10 \text{ ns}$$

## See also

[P2\\_Digin\\_FIFO\\_Clear](#), [P2\\_Digin\\_FIFO\\_Enable](#), [P2\\_Digin\\_FIFO\\_Full](#), [P2\\_Digin\\_FIFO\\_Read\\_Timer](#), [P2\\_Digin\\_Edge](#)

## Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, OPT-16 Rev. E

## P2\_Digin\_FIFO\_Read

## Example

```
#Include ADwinPro_All.Inc

Dim Data_1[10000], Data_2[10000] As Long
Dim num, index As Long

Init:
    P2_Digprog(1,1100b)          'channels 0:15 as inputs
    P2_Digin_FIFO_Enable(1,0)'output edge edge detection off
    P2_Digin_FIFO_Clear(1)      'clear FIFO
    P2_Digin_FIFO_Enable(1,10011b)'edge detection channels 1,2,5
    index = 1

Event:
    num = P2_Digin_FIFO_Full(1)'number of value pairs
    If (num>50) Then
        Rem read value pairs
        P2_Digin_FIFO_Read(1, num, Data_1, Data_2, index)
        index=index + num
        If (index>10000) Then index = 1
    EndIf
```

**P2\_Digin\_Fifo\_Read\_Fast** reads the value pairs from the FIFO of the edge detection unit and writes them into a single array.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Digin_Fifo_Read_Fast(module, count, valuepairs[],
    start_index)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Number (1...511) of value pairs to be read.	LONG
<b>valuepairs[]</b>	Array where the value pairs are written to, alternately a level status bit pattern and a time stamp. Each level status bit corresponds to a digital input (see table below).	LONG ARRAY
<b>start_index</b>	Start index for both arrays <b>valuepairs[]</b> and <b>timestamp[]</b> , where the first value are written.	LONG

Bit no.	31	30	29	...	2	1	0
Input	31	30	29	...	2	1	0

## Notes

The function corresponds to **P2\_Digin\_Fifo\_Read** but is faster.

No more value pairs may be read than are saved in the FIFO. Thus, before reading there must be a check with **P2\_Digin\_FIFO\_Full**, how much value pairs are saved in the FIFO.

The arrays must be dimensioned with sufficient size, so that all value pairs may be written into the arrays.

The array **valuepairs[]** contains value pairs of level status and appropriate time stamp:

- One array element holds the level status of output channels 0...31 as bit pattern.
- The next array element hold a time stamp (absolute or relative, see **P2\_Dig\_FIFO\_Mode**).

The time difference between 2 level status patterns is the difference of the appropriate time stamps, measured in units of 10ns:

$$\Delta t = (\text{stamp}_1 - \text{stamp}_2) \cdot 10 \text{ ns}$$

## See also

[P2\\_Digin\\_FIFO\\_Clear](#), [P2\\_Digin\\_FIFO\\_Enable](#), [P2\\_Digin\\_FIFO\\_Full](#), [P2\\_Digin\\_FIFO\\_Read\\_Timer](#), [P2\\_Digin\\_Edge](#)

## Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, OPT-16 Rev. E

## P2\_Digin\_Fifo\_Read\_Fast

## Example

```
#Include ADwinPro_All.inc

Dim Data_1[20000] As Long
Dim num, index As Long

Init:
    P2_Digprog(1,1100b)          'channels 0:15 as inputs
    P2_Digin_FIFO_Enable(1,0)'output edge edge detection off
    P2_Digin_FIFO_Clear(1)      'clear FIFO
    P2_Digin_FIFO_Enable(1,10011b)'edge detection channels 1,2,5
    index = 1

Event:
    num = P2_Digin_FIFO_Full(1)'number of value pairs
    If (num > 50) Then
        Rem read value pairs
        P2_Digin_Fifo_Read_Fast(1, num, Data_1, index)
        index = index + num
        If (index > 10000) Then index = 1
    EndIf
```

**P2\_Digin\_FIFO\_Read\_Timer** returns the current status of the 100MHz timer on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Digin_FIFO_Read_Timer(module)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Current value ( $-2^{31}-1 \dots 2^{31}$ ) of the 100MHz timer.	LONG

## Notes

The module timer is used to provide time stamps for the edge detection unit, see **P2\_Digin\_FIFO\_Enable**.

The timer value is increased every 10ns by 1, so the timer will reach the original timer value after about 43 seconds ( $= 10\text{ns} \times 2^{32}$ ). For comparison of time this "overflow" must be considered, so the timer value must be queried regularly in the program before a overflow has happened.

## See also

[P2\\_Digin\\_FIFO\\_Enable](#), [P2\\_Digin\\_FIFO\\_Read](#)

## Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, OPT-16 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Dim Data_1[10000], Data_2[10000] As Long
Dim num, index As Long

Init:
  P2_Digprog(1,1100b)      'channels 0:15 as inputs
  P2_Digin_FIFO_Enable(1,0)'output edge edge detection off
  P2_Digin_FIFO_Clear(1)   'clear FIFO
  P2_Digin_FIFO_Enable(1,10011b)'edge detection channels 1,2,5
  index = 1

Event:
  num = P2_Digin_FIFO_Full(1)'number of value pairs
  If (num>50) Then
    Rem read value pairs
    P2_Digin_FIFO_Read(1, num, Data_1, Data_2, index)
    index=index + num
    If (index>10000) Then index = 1
  EndIf
```

## P2\_Digin\_FIFO\_Read\_Timer

## P2\_Digin\_Long

**P2\_Digin\_Long** returns the status of the inputs (bits 31...00) of the specified module as bit pattern.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Digin_Long(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Bit pattern. Each bit (31...0) corresponds to the input status of a digital input (see table). Bit = 0: Input has low level. Bit = 1: Input has high level.	LONG

Bit no.	31	30	29	...	2	1	0
Input	31	30	29	...	2	1	0

### Notes

It is recommended to first program the specified channels as inputs using **P2\_Digprog**.

### See also

[P2\\_Dig\\_Latch](#), [P2\\_Digprog](#), [P2\\_Digout\\_Long](#)

### Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, OPT-16 Rev. E

### Example

```
#Include ADwinPro_All.Inc

Init:
P2_Digprog(1,0)           'Set DIO31:00 as inputs

Event:
Par_1 = P2_Digin_Long(1) 'Read all inputs
```

**P2\_Digout** sets a single output of the specified module to the level "high" or "low". All other outputs remain unchanged.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Digout (module, output, value)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>output</b>	Number of the output to be set (0...31).	LONG
<b>value</b>	New status of the selected output: 0: Low level. 1: High level.	LONG

## Notes

The specified channels must be first programmed as outputs using **P2\_Digprog**.

You can set or clear any output without changing the status of the remaining outputs.

With module version TRA-16-G Rev. E, high level switches to ground, not to V<sub>CC</sub>.

## See also

[P2\\_Digout\\_Long](#), [P2\\_Digout\\_Bits](#), [P2\\_Digprog](#)

## Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, REL-16 Rev. E, TRA-16 Rev. E

## Example

```
#Include ADwinPro_All.Inc
```

### Init:

```
REM Set channels 0...15 as inputs, 16...31 as outputs
P2_Digprog(1,01100b)
```

### Event:

```
If (P2_Digin_Long(1) And 8000h = 1) Then
    P2_Digout(1,31,0)      'channel 15 is set: clear bit 31
Else
    P2_Digout(1,31,1)      'channel 15 is cleared: set bit 31
EndIf
```

## P2\_Digout

## P2\_Digout\_Bits

**P2\_Digout\_Bits** sets the specified outputs of the specified module to the levels "high" or "low".

### Syntax

```
#Include ADwinPro_All.Inc

P2_Digout_Bits(module, set, clear)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>set</b>	Bit pattern that sets specified digital outputs to the level "high". Bit = 0: Output remains unchanged. Bit = 1: Set output to level "high".	LONG
<b>clear</b>	Bit pattern that sets specified digital outputs to the level "low". Bit = 0: Output remains unchanged. Bit = 1: Set output to level "low".	LONG

Bit no.	31	30	...	2	1	0
Output	31	30	...	2	1	0

### Notes

The specified channels must be first programmed as outputs using **P2\_Digprog**.

You can set or clear any required outputs without changing the status of the remaining outputs.

For clarity reasons please note that the bits in **set** must not be set in the bit pattern **clear** at the same time, and vice versa.

With module version TRA-16-G Rev. E, high level switches to ground, not to V<sub>CC</sub>.

### See also

[P2\\_Dig\\_Latch](#), [P2\\_Digout\\_Reset](#), [P2\\_Digout\\_Set](#), [P2\\_Digprog](#), [P2\\_Digout\\_Long](#)

### Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, REL-16 Rev. E, TRA-16 Rev. E

### Example

```
#Include ADwinPro_All.Inc

Init:
    REM Set channels 0...31 as outputs
    P2_Digprog(1, 01111b)

Event:
    If (Par_1 = 1) Then          'Get condition
        REM lower word: Set byte MSBs, clear all other bits
        P2_Digout_Bits(1, 8080h, 7F7Fh)
    Else
        REM lower word: set odd-numbered bits, clear even-numbered
        P2_Digout_Bits(1, 5555h, 0AAAhh)
    EndIf
```



**P2\_Digout\_FIFO\_Clear** stops the edge output and clears the edge output FIFO on the specified module.

## Syntax

```
#Include ADwinPro_All.inc  
  
P2_Digout_FIFO_Clear ( module )
```

## Parameters

**module** Specified module address (1...15). LONG

## Notes

Before first use, the FIFO must be cleared. Then, the FIFO can be filled with data using **P2\_Digout\_FIFO\_Write**.

If the edge output has been stopped with **P2\_Digout\_FIFO\_Clear**, it can only be started with **P2\_Digout\_FIFO\_Start** again.

## See also

[P2\\_Digout\\_FIFO\\_Enable](#), [P2\\_Dig\\_FIFO\\_Mode](#), [P2\\_Digout\\_FIFO\\_Start](#), [P2\\_Digout\\_FIFO\\_Write](#), [P2\\_Digprog](#)

## Valid for

DIO-32-TiCo Rev. E03

## Example

see [P2\\_Dig\\_FIFO\\_Mode](#)

## P2\_Digout\_FIFO\_Clear

## P2\_Digout\_FIFO\_Empty

**P2\_Digout\_FIFO\_Empty** returns the number of free value pairs in the edge output FIFO.

### Syntax

```
#Include ADwinPro_All.inc  
  
ret_value = P2_Digout_FIFO_Empty(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	<a href="#">LONG</a>
<b>ret_value</b>	Number (0...511) of free value pairs in the FIFO.	<a href="#">LONG</a>

### Notes

The FIFO array can hold up to 511 value pairs (level status and time stamp).

### See also

[P2\\_Dig\\_FIFO\\_Mode](#), [P2\\_Digout\\_FIFO\\_Read\\_Timer](#), [P2\\_Digout\\_FIFO\\_Start](#), [P2\\_Digout\\_FIFO\\_Write](#)

### Valid for

DIO-32-TiCo Rev. E03

### Example

see [P2\\_Dig\\_FIFO\\_Mode](#)

**P2\_Digout\_FIFO\_Enable** sets the output channels of the specified module where edges are output.

## Syntax

```
#Include ADwinPro_All.inc
```

```
P2_Digout_FIFO_Enable (module, channels)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>channels</b>	Bitt pattern to set the selected output channels.	LONG

Bit no.	31	30	...	2	1	0
Ouptu	31	30	...	2	1	0

## Notes

Edges can only be output to output channels. The specified channels must be first programmed as outputs using **P2\_Digprog**.

The FIFO must be enabled for edge output with **P2\_Dig\_FIFO\_Mode**.

The edge output sets edges only on the output channels selected with **P2\_Digout\_FIFO\_Enable**. You can set the levels of the other output channels—and only of these—with instructions like **P2\_Digout\_Long**.

The levels and points of time of egde ouput are set with **P2\_Digout\_FIFO\_Write**.

## See also

[P2\\_Digout\\_FIFO\\_Clear](#), [P2\\_Dig\\_FIFO\\_Mode](#), [P2\\_Digout\\_FIFO\\_Start](#), [P2\\_Digout\\_FIFO\\_Write](#), [P2\\_Digprog](#), [P2\\_Digout\\_Long](#)

## Valid for

DIO-32-TiCo Rev. E03

## Example

see [P2\\_Dig\\_FIFO\\_Mode](#)

## P2\_Digout\_FIFO\_Enable

## P2\_Digout\_FIFO\_Read\_Timer

**P2\_Digout\_FIFO\_Read\_Timer** returns the current value of the 100MHz counter on the specified module.

### Syntax

```
#Include ADwinPro_All.inc  
  
ret_val = P2_Digout_FIFO_Read_Timer(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Current value ( $-2^{31}-1 \dots 2^{31}$ ) of the 100MHz counter.	LONG

### Notes

The module counter is used for exact edge output timing at predefined points of time, see **P2\_Digout\_FIFO\_Write**.

The counter value can only be used in the FIFO operation mode with absolute time values, i.e. parameter **mode** = 1 in **P2\_Dig\_FIFO\_Mode**

The counter is increased every 10ns by 1, so the counter reaches the previous value again after about 43 seconds ( $= 10\text{ns} \times 2^{32}$ ). A "missed" edge output is done only after the timer has run once around.

### See also

[P2\\_Dig\\_FIFO\\_Mode](#), [P2\\_Digout\\_FIFO\\_Empty](#), [P2\\_Digout\\_FIFO\\_Start](#), [P2\\_Digout\\_FIFO\\_Write](#), [P2\\_Digprog](#)

### Valid for

DIO-32-TiCo Rev. E03

### Example

- / -

**P2\_Digout\_FIFO\_Start** starts the edge output on the specified module.

## Syntax

```
#Include ADwinPro_All.inc

P2_Digout_FIFO_Start(module_pattern)
```

## Parameters

**module\_** Bit pattern to access the modules: LONG  
**pattern** Bit = 0: Ignore module.  
 Bit = 1: Start edge output on the module.

Bits in <b>module_pattern</b>	31:15	14	13	...	01	00
Module address	–	15	14	...	2	1

## Notes

After start, the module counter starts to count with 0. The module counter is used to do exact output timing, see **P2\_Digout\_FIFO\_Write**.

The counter is increased every 10ns by 1, so the counter reaches the previous value again after about 43 seconds ( $= 10\text{ns} \times 2^{32}$ ). A "missed" edge output is done only after the timer has run once around.

## See also

[P2\\_Digout\\_FIFO\\_Clear](#), [P2\\_Digout\\_FIFO\\_Enable](#), [P2\\_Dig\\_FIFO\\_Mode](#), [P2\\_Digout\\_FIFO\\_Read\\_Timer](#), [P2\\_Digout\\_FIFO\\_Write](#), [P2\\_Digprog](#)

## Valid for

DIO-32-TiCo Rev. E03

## Example

see [P2\\_Dig\\_FIFO\\_Mode](#)

## P2\_Digout\_FIFO\_Start

## P2\_Digout\_FIFO\_Write

**P2\_Digout\_FIFO\_Write** writes value pairs into the output edge FIFO.

### Syntax

```
#Include ADwinPro_All.inc
```

```
P2_Digout_FIFO_Write(module, count, value[],  
start_index)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>count</b>	Number (1...511) of value pairs to write.	LONG
<b>value[ ]</b>	Array containing bit patterns of level status and time stamps for output timing. Each bit corresponds to a digital output (see table).	LONG ARRAY
<b>start_index</b>	Start index of the array <b>value[ ]</b> , where the first value is read.	LONG

Bit no.	31	30	...	2	1	0
Output	31	30	...	2	1	0

### Notes

You must not write more value pairs into the FIFO than are free.

The FIFO array may contain 511 value pairs (level status and time stamp) in maximum. If and as long as the FIFO array is filled completely, no more value pair can be written into.

The array **value[ ]** has to contain value pairs of level status and appropriate time stamp:

- Odd numbered array elements hold the level status of output channels 0...31 as bit pattern.
- Even numbered array elements hold a time stamp (absolute or relative, see **P2\_Dig\_FIFO\_Mode**). The difference between two output times must be at least 20ns.

The output runs like this:

- The module counter is increased every 10ns by 1.
- If the counter value equals the time stamp of the current value pair in the FIFO, the bit pattern are output to the specified output channels.
- If a bit pattern has been output, the value pair is deleted from the FIFO.
- The value pairs are processed in the order as they were written in to the FIFO.

Therefore:

A time stamp defines the exact output time, and in time units of 10ns.

The value can be given in two ways:

- As absolute value in relation to the starting time of the module counter using **P2\_Digout\_FIFO\_Start**.  
A time stamp of 153 will have the appropriate bit pattern be output exactly at 1.53µs after the module counter has started.
- As relative value which is relative to the previous time stamp.  
A time stamp of 153 will have the appropriate bit pattern be output exactly at 1.53µs after the previous pattern was output.

Time stamps must be stored in ascending order.

The FIFO must be filled with data early enough, so that the next output time is located in the future. In detail:

- With absolute values the time stamp must be greater than the current timer value. Otherwise the edge output is "missed" and executed only after the timer has run once around. (about 43 seconds).
- With relative values the time stamp must be greater than the time period since the previous pattern output. If this fails, the bit pattern is output immediately (but with delay); the next time stamp will then be relative to the delayed output time.

## See also

[P2\\_Digout\\_FIFO\\_Empty](#), [P2\\_Digout\\_FIFO\\_Enable](#), [P2\\_Dig\\_FIFO\\_Mode](#), [P2\\_Digout\\_FIFO\\_Read\\_Timer](#), [P2\\_Digout\\_FIFO\\_Start](#), [P2\\_Digout\\_Long](#), [P2\\_Digprog](#)

## Valid for

DIO-32-TiCo Rev. E03

## Example

see [P2\\_Dig\\_FIFO\\_Mode](#)

## P2\_Digout\_Long

**P2\_Digout\_Long** sets or clears all outputs on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Digout_Long(module,pattern)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern that sets the digital outputs: Bit = 0: Set output to level "low". Bit = 1: Set output to level "high".	LONG

Bit no.	31	30	...	2	1	0
Output	31	30	...	2	1	0

### Notes

The specified channels must be first programmed as outputs using **P2\_Digprog**.

With module version TRA-16-G Rev. E, high level switches to ground, not to V<sub>CC</sub>.

### See also

[P2\\_Digout](#), [P2\\_Digout\\_Bits](#), [P2\\_Digprog](#)

### Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, REL-16 Rev. E, TRA-16 Rev. E

### Example

```
#Include ADwinPro_All.Inc

Init:
    P2_Digprog(1,0FFFFh)      'DIO31:00 as outputs

Event:
    P2_Digout_Long(1,1000000)'Output the value 1 million as binary
                                'value on the DIOS
```



**P2\_Digout\_Reset** sets the specified outputs of the specified module to the level "low".

## Syntax

```
#Include ADwinPro_All.Inc

P2_Digout_Reset (module, clear)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>clear</b>	Bit pattern that sets specified digital outputs to the level "low". Bit = 0: Output remains unchanged. Bit = 1: Set output to level "low".	LONG

Bit no.	31	30	...	2	1	0
Output	31	30	...	2	1	0

## Notes

The specified channels must be first programmed as outputs using **P2\_Digprog**.

You can clear any required outputs without changing the status of the remaining outputs.

With module version TRA-16-G Rev. E, high level switches to ground, not to  $V_{CC}$ .

## See also

[P2\\_Digout](#), [P2\\_Digout\\_Bits](#), [P2\\_Digout\\_Long](#), [P2\\_Digout\\_Set](#), [P2\\_Digprog](#)

## Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, REL-16 Rev. E, TRA-16 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Init:
    REM Set channels 0...31 as outputs
    P2_Digprog(1, 01111b)

Event:
    If (Par_1 = 1) Then          'Get condition
        REM lower word: clear even-numbered bits
        P2_Digout_Reset(1, 0AAAAh)
    EndIf
```

## P2\_Digout\_Reset

## P2\_Digout\_Set

**P2\_Digout\_Set** sets the specified outputs of the specified module to the level "high".

### Syntax

```
#Include ADwinPro_All.Inc

P2_Digout_Set(module, set)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>set</b>	Bit pattern that sets specified digital outputs to the level "high". Bit = 0: Output remains unchanged. Bit = 1: Set output to level "high".	LONG

Bit no.	31	30	...	2	1	0
Output	31	30	...	2	1	0

### Notes

The specified channels must be first programmed as outputs using **P2\_Digprog**.

You can set any required outputs without changing the status of the remaining outputs.

With module version TRA-16-G Rev. E, high level switches to ground, not to V<sub>CC</sub>.

### See also

[P2\\_Digout](#), [P2\\_Digout\\_Bits](#), [P2\\_Digout\\_Long](#), [P2\\_Digout\\_Reset](#), [P2\\_Digprog](#)

### Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, REL-16 Rev. E, TRA-16 Rev. E

### Example

```
#Include ADwinPro_All.Inc
```

#### Init:

```
REM Set channels 0...31 as outputs
P2_Digprog(1,1111b)
```

#### Event:

```
If (Par_1 = 1) Then          'Get condition
REM lower word: Set byte MSBs
P2_Digout_Set(1,8080h)
EndIf
```

**P2\_Digprog** programs the digital channels 0...31 of the specified module as inputs or outputs in groups of 8.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Digprog(module,pattern)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern that sets the channels as inputs or outputs: Bit = 0: Set channel as input. Bit = 1: Set channel as output.	LONG

Bit no.	31...4	3	2	1	0
channel no.	–	31:24	23:16	15:08	07:00

## Notes

After power-up of the system all channels are configured as inputs.

Channels can only be set as inputs or outputs in groups of 8, 4 relevant bits only, the other bits are ignored.

## See also

[P2\\_Dig\\_Latch](#), [P2\\_Dig\\_Read\\_Latch](#), [P2\\_Dig\\_Write\\_Latch](#)

[P2\\_Digin\\_Long](#), [P2\\_Digout\\_Bits](#), [P2\\_Digout](#), [P2\\_Digout\\_Long](#), [P2\\_Get\\_Digout\\_Long](#)

## Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E

## Example

```
#Include ADwinPro_All.Inc
```

### Init:

```
REM Configure channels 0...7 of the DIO module no. 1 as inputs and
REM channels 8...31 as inputs
P2_Digprog(1, 1110b)
```

## P2\_Digprog

## P2\_Get\_Digout\_Long

**P2\_Get\_Digout\_Long** returns the contents of the output latch (register for digital outputs) on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
ret_val = P2_Get_Digout_Long(module)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>ret_val</b>	Contents of the output latch (bits 31:00).	LONG

### Notes

Returning the current status of the outputs instead of the output latch is technically impossible.

With module version TRA-16-G Rev. E, high level switches to ground, not to  $V_{CC}$ .

### See also

[P2\\_Dig\\_Latch](#), [P2\\_Dig\\_Read\\_Latch](#), [P2\\_Dig\\_Write\\_Latch](#)

[P2\\_Digprog](#), [P2\\_Digin\\_Long](#), [P2\\_Digout\\_Bits](#), [P2\\_Digout](#), [P2\\_Digout\\_Long](#)

### Valid for

DIO-32 Rev. E, DIO-32-TiCo Rev. E, REL-16 Rev. E, TRA-16 Rev. E

### Example

```
#Include ADwinPro_All.Inc
```

#### Event:

```
Par_1 = P2_Get_Digout_Long(1)'return bits 31:00 from the latch
```

### 3.6 Pro II: Counter Modules

This section describes instructions which apply to Pro II counter modules:

- [P2\\_Cnt\\_Clear](#) (page 162)
- [P2\\_Cnt\\_Enable](#) (page 163)
- [P2\\_Cnt\\_Get\\_Status](#) (page 164)
- [P2\\_Cnt\\_Get\\_PW](#) (page 165)
- [P2\\_Cnt\\_Get\\_PW\\_HL](#) (page 166)
- [P2\\_Cnt\\_Latch](#) (page 167)
- [P2\\_Cnt\\_Mode](#) (page 168)
- [P2\\_Cnt\\_PW\\_Enable](#) (page 170)
- [P2\\_Cnt\\_PW\\_Latch](#) (page 171)
- [P2\\_Cnt\\_Read](#) (page 172)
- [P2\\_Cnt\\_Read4](#) (page 173)
- [P2\\_Cnt\\_Read4](#) (page 173)
- [P2\\_Cnt\\_Read\\_Latch](#) (page 175)
- [P2\\_Cnt\\_Read\\_Latch4](#) (page 176)
- [P2\\_Cnt\\_Sync\\_Latch](#) (page 177)
- [P2\\_SSI\\_Mode](#) (page 179)
- [P2\\_SSI\\_Read](#) (page 180)
- [P2\\_SSI\\_Read2](#) (page 182)
- [P2\\_SSI\\_Set\\_Bits](#) (page 183)
- [P2\\_SSI\\_Set\\_Clock](#) (page 184)
- [P2\\_SSI\\_Set\\_Delay](#) (page 185)
- [P2\\_SSI\\_Start](#) (page 186)
- [P2\\_SSI\\_Status](#) (page 187)

In the [Instruction List sorted by Module Types](#) (annex A.2) you will find which of the functions corresponds to the *ADwin-Pro II* modules.

It is presumed that application examples use the module address 1.



## P2\_Cnt\_Clear

**P2\_Cnt\_Clear** sets the counter values of one or more counters to 0 (zero), according to a given bit **pattern**.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Cnt_Clear(module, pattern)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern, assignment to the counters, see table. Bit = 0: No function. Bit = 1: Reset counter.	LONG

Bit no.	31...4	3	2	1	0
Counter no.	–	4	3	2	1

### Notes

After **P2\_Cnt\_Clear** has been executed the bit pattern is automatically reset to 0 (zero), so the counters start counting from 0.

### See also

[P2\\_Cnt\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#), [P2\\_Cnt\\_Latch](#), [P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_Read](#), [P2\\_Cnt\\_Read4](#), [P2\\_Cnt\\_Read\\_Latch](#), [P2\\_Cnt\\_Read\\_Latch4](#)

### Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim old_1, new_1 As Long
Dim old_2, new_2 As Long

Init:
    old_1 = 0 'initialize
    old_2 = 0
    Rem Counter 1: clock-dir, enable CLR input
    P2_Cnt_Mode(1,1,10000b)
    P2_Cnt_Mode(1,2,0) 'all counters use external
    trigger
    P2_Cnt_Clear(1,11b) 'reset counters 1+2 to 0
    P2_Cnt_Enable(1,11b) 'start counters 1+2, stop
    counters 3+4

Event:
    P2_Cnt_Latch(1,11b) 'latch both counters 1+2
    new_1 = P2_Cnt_Read_Latch(1,1) 'read latch A counter 1 and...
    new_2 = P2_Cnt_Read_Latch(1,2) 'latch A counter 2
    Par_1 = new_1 - old_1 'get difference (f = impulses / time)
    Par_2 = new_2 - old_2 '---
    old_1 = new_1 'store new counter value as old
    old_2 = new_2 '---'
```

**P2\_Cnt\_Enable** enables or disables the counters selected by **pattern**.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Cnt_Enable(module, pattern)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern Bit = 0: Disable counter. Bit = 1: Enable counter.	LONG

Bit no.	31...4	3	2	1	0
Counter no.	–	VR4	VR3	VR2	VR1

## Notes

PWM counters are started or stopped with **P2\_Cnt\_PW\_Enable**.

## See also

[P2\\_Cnt\\_Clear](#), [P2\\_Cnt\\_PW\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#), [P2\\_Cnt\\_Latch](#), [P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_Read](#), [P2\\_Cnt\\_Read4](#), [P2\\_Cnt\\_Read\\_Latch](#), [P2\\_Cnt\\_Read\\_Latch4](#)

## Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim old_1, new_1 As Long
Dim old_2, new_2 As Long

Init:
    old_1 = 0 'initialize...
    old_2 = 0
    Rem Counter 1: clock-dir, enable CLR input
    P2_Cnt_Mode(1,1,10000b)
    P2_Cnt_Mode(1,2,0) 'all counters use external
trigger
    P2_Cnt_Clear(1,11b) 'reset counters 1+2 to 0
    P2_Cnt_Enable(1,11b) 'start counters 1+2, stop
counters 3+4

Event:
    P2_Cnt_Latch(1,11b) 'latch both counters 1+2
    new_1 = P2_Cnt_Read_Latch(1,1)'read latch A counter 1 and...
    new_2 = P2_Cnt_Read_Latch(1,2)'latch A counter 2
    Par_1 = new_1 - old_1'get difference (f = impulses / time)
    Par_2 = new_2 - old_2'--
    old_1 = new_1 'store new counter value as old
    old_2 = new_2 '--
```

## P2\_Cnt\_Enable

## P2\_Cnt\_Get\_Status

**P2\_Cnt\_Get\_Status** returns the counter status register of one counter.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Cnt_Get_Status(module, counter_no)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>counter_no</b>	Counter number: 1...4.	LONG
<b>ret_val</b>	Content of the counter status register: Hints for potential error sources. Meaning of bits 0...4 see table.	LONG

Bit no.	31...5	4	3	2	1	0
Signal	–	C	L	N	B	A

- : don't care (mask with **01Fh**).

A: Signal A (static).

B: Signal B(static).

N: CLR/Latch input (static).

L: Line error (cable not connected or the line is broken).

C: Correlation error\* (signals A and B are identical, they are not phase-shifted by approx. 90°).

### Notes

A line error (L) can only be detected with differential inputs! For TTL inputs these bits are always 0.

The status register is automatically reset by reading.

### See also

[P2\\_Cnt\\_Enable](#), [P2\\_Cnt\\_PW\\_Enable](#), [P2\\_Cnt\\_Get\\_PW](#), [P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_Read](#)

### Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

### Example

- / -



**P2\_Cnt\_Get\_PW** returns frequency and duty cycle of a PWM counter.

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_Cnt_Get_PW(module, pwm_no, frequency, dutycycle)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pwm_no</b>	Number (1...4) of PWM counter.	LONG
<b>frequency</b>	Frequency in Hertz: 0.023 Hz ...100MHz.	FLOAT
		CONST
<b>dutycycle</b>	Duty cycle in percent: 0.0...100.0.	FLOAT
		CONST

## Notes

The return values are given in the parameters **frequency** and **dutycycle**.

## See also

[P2\\_Cnt\\_PW\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#), [P2\\_Cnt\\_Get\\_PW\\_HL](#), [P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_PW\\_Latch](#)

## Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim old_1, new_1 As Long
Dim old_2, new_2 As Long

Init:
    old_1 = 0                'initialize
    old_2 = 0
    P2_Cnt_Mode(1,1,0)       'Counter 1: PWM at input A
    P2_Cnt_Mode(1,2,0)       'Counter 2: PWM at input A
    P2_Cnt_PW_Enable(1,0011b) 'start PWM counters 1+2, stop 3+4

Event:
    P2_Cnt_PW_Latch(1,11b)   'latch both counters 1+2
    P2_Cnt_Get_PW_HL(1,1,Par_1,Par_2) 'read high/low time
    P2_Cnt_Get_PW(1,1,FPar_1,FPar_2) 'read frequency / duty
    cycle
```

## P2\_Cnt\_Get\_PW

## P2\_Cnt\_Get\_PW\_HL

**P2\_Cnt\_Get\_PW\_HL** returns a stored high and low time of a PWM counter.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Cnt_Get_PW_HL(module, counter_
no, hightime, lowtime)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>counter_</b> <b>no</b>	Number (1...4) of PWM counter.	LONG
<b>hightime</b>	Pulse duration in units of 10ns: high level time of PWM signal.	LONG CONST
<b>lowtime</b>	Pulse period in units of 10ns:.0): low level time of PWM signal.	LONG CONST

### Notes

The return values are given in the parameters **hightime** and **lowtime**.

### See also

[P2\\_Cnt\\_PW\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#), [P2\\_Cnt\\_Get\\_PW](#), [P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_PW\\_Latch](#)

### Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim old_1, new_1 As Long
Dim old_2, new_2 As Long

Init:
    old_1 = 0                'initialize
    old_2 = 0
    P2_Cnt_Mode(1,1,0)       'Counter 1: PWM at input A
    P2_Cnt_Mode(1,2,0)       'Counter 2: PWM at input A
    P2_Cnt_PW_Enable(1,0011b) 'start PWM counters 1+2, stop 3+4

Event:
    P2_Cnt_PW_Latch(1,11b)   'latch both counters 1+2
    P2_Cnt_Get_PW_HL(1,1,Par_1,Par_2) 'read high/low time
    P2_Cnt_Get_PW(1,1,FPar_1,FPar_2) 'read frequency / duty
    cycle
```

**P2\_Cnt\_Latch** transfers the current counter values of one or more counters into the relevant Latch A, depending on the bit **pattern**.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Cnt_Latch(module, pattern)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern. Bit = 0: no function. Bit = 1: transfer counter values into Latch A .	LONG

Bit no.	31...4	3	2	1	0
Counter no.	–	4	3	2	1

## Notes

After **Cnt\_Latch** has been executed the bit pattern is automatically reset to 0 (zero).

Latch A is read out into a variable with **Cnt\_Read\_Latch** command.

## See also

[P2\\_Cnt\\_Clear](#), [P2\\_Cnt\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#), [P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_Read](#), [P2\\_Cnt\\_Read4](#), [P2\\_Cnt\\_Read\\_Latch](#), [P2\\_Cnt\\_Read\\_Latch4](#)

## Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim old_1, new_1 As Long
Dim old_2, new_2 As Long

Init:
    old_1 = 0                'initialize
    old_2 = 0
    Rem Counter 1: clock-dir, enable CLR input
    P2_Cnt_Mode(1,1,10000b)
    P2_Cnt_Mode(1,2,0)      'all counters use external
    trigger
    P2_Cnt_Clear(1,11b)     'reset counters 1+2 to 0
    P2_Cnt_Enable(1,11b)   'start counters 1+2, stop
    counters 3+4

Event:
    P2_Cnt_Latch(1,11b)    'latch both counters 1+2
    new_1 = P2_Cnt_Read_Latch(1,1)'read latch A counter 1 and...
    new_2 = P2_Cnt_Read_Latch(1,2)'latch A counter 2
    Par_1 = new_1 - old_1 'get difference (f = impulses / time)
    Par_2 = new_2 - old_2 '---
    old_1 = new_1         'store new counter value as old
    old_2 = new_2         '---'
```

## P2\_Cnt\_Latch

## P2\_Cnt\_Mode

**P2\_Cnt\_Mode** defines the operating mode of one counter.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Cnt_Mode(module, counter_no, pattern)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>counter_no</b>	Counter number: 1...4.	LONG
<b>pattern</b>	Bit pattern to set the operating mode of a counter.	LONG

Bit no.	Meaning
Bit 0	Counter mode: Bit = 0: mode clock/direction. Bit = 1: mode A-B.
Bit 1	Clear mode. Signal condition which clears the counter: Bit = 0: TTL level high at input CLR. Bit = 1: TTL level high at all inputs A, B, CLR. Available in mode A-B only.
Bit 2	Revert input A / CLK in mode clock/direction: Bit = 0: Input is not reverted. Bit = 1: Input is reverted.
Bit 3	Revert input B / DIR in mode clock/direction: Bit = 0: Input is not reverted. Bit = 1: Input is reverted.
Bit 4	Set use of input CLR / LATCH. Bit = 0: CLR input: clear counter. Bit = 1: LATCH input: latch counter.
Bit 5	Enable input CLR / LATCH. Bit = 0: Input CLR / LATCH is disabled. Bit = 1: Input CLR / LATCH is enabled.
Bit 6	Select edge for PWM analysis. Bit = 0: rising edge. Bit = 1: falling edge.
Bit 7,8	Select input for PWM analysis. 00b: Input A / CLK 01b: Input B / DIR 10b: Input CLR / LATCH
Bits 9...31	reserved

### Notes

Please use **P2\_Cnt\_Mode** only when the counter is disabled, see **P2\_Cnt\_Enable**.

With standard clear mode (bit 1=0), the counter value is reset to zero as long as TTL level high is given at the input. In order to clear the counter, the input CLR must be enabled with bit 5=1.

### See also

[P2\\_Cnt\\_Clear](#), [P2\\_Cnt\\_Enable](#), [P2\\_Cnt\\_PW\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#)

## Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim old_1, new_1 As Long
Dim old_2, new_2 As Long

Init:
    old_1 = 0                'initialize
    old_2 = 0
    Rem Counter 1: clock-dir, enable CLR input
    P2_Cnt_Mode(1,1,10000b)
    P2_Cnt_Mode(1,2,0)      'all counters use external
trigger
    P2_Cnt_Clear(1,11b)     'reset counters 1+2 to 0
    P2_Cnt_Enable(1,11b)    'start counters 1+2, stop
counters 3+4

Event:
    P2_Cnt_Latch(1,11b)     'latch both counters 1+2
    new_1 = P2_Cnt_Read_Latch(1,1)'read latch A counter 1 and...
    new_2 = P2_Cnt_Read_Latch(1,2)'latch A counter 2
    Par_1 = new_1 - old_1 'get difference (f = impulses / time)
    Par_2 = new_2 - old_2 '-"-
    old_1 = new_1          'store new counter value as old
    old_2 = new_2          '-"-
```

## P2\_Cnt\_PW\_Enable

**P2\_Cnt\_PW\_Enable** enables or disables the counters selected by **pattern**.

### Syntax

```
#Include ADwinPro_All.Inc

P2_Cnt_PW_Enable(module, pattern)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern Bit = 0: Disable counter. Bit = 1: Enable counter.	LONG

Bit no.	31...4	3	2	1	0
Counter no.	–	PW 4	PW 3	PW 2	PW 1

### Notes

Standard counters are started or stopped with **P2\_Cnt\_Enable**.

### See also

[P2\\_Cnt\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#), [P2\\_Cnt\\_Get\\_PW](#), [P2\\_Cnt\\_Get\\_PW\\_HL](#), [P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_PW\\_Latch](#)

### Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim old_1, new_1 As Long
Dim old_2, new_2 As Long

Init:
    old_1 = 0 'initialize...
    old_2 = 0
    Rem Counter 1: clock-dir, enable CLR input
    P2_Cnt_Mode(1,1,10000b)
    P2_Cnt_Mode(1,2,0) 'all counters use external trigger
    P2_Cnt_Clear(1,11b) 'reset counters 1+2 to 0
    P2_Cnt_PW_Enable(1,0011b) 'start PWM counters 1+2, stop 3+4

Event:
    P2_Cnt_PW_Latch(1,11b) 'latch both counters 1+2
    P2_Cnt_Get_PW_HL(1,1,Par_1,Par_2) 'read high/low time
    P2_Cnt_Get_PW(1,1,FPar_1,FPar_2) 'read frequency / duty cycle
```

**P2\_Cnt\_PW\_Latch** copies the value of one or more PWM counters into a buffer.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Cnt_PW_Latch(module, pattern)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern. Bit = 0: no function. Bit = 1: transfer PWM counter value into a buffer.	LONG

Bit no.	31...4	3	2	1	0
Counter no.	–	4	3	2	1

## Notes

The buffer is to be read with **Cnt\_Get\_PW** or **Cnt\_Get\_PW\_HL**.

## See also

[P2\\_Cnt\\_PW\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#), [P2\\_Cnt\\_Get\\_PW](#), [P2\\_Cnt\\_Get\\_PW\\_HL](#), [P2\\_Cnt\\_Mode](#)

## Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim old_1, new_1 As Long
Dim old_2, new_2 As Long

Init:
    old_1 = 0                'initialize
    old_2 = 0
    P2_Cnt_Mode(1,1,0)      'Counter 1: PWM at input A
    P2_Cnt_Mode(1,2,0)      'Counter 2: PWM at input A
    P2_Cnt_PW_Enable(1,0011b) 'start PWM counters 1+2, stop 3+4

Event:
    P2_Cnt_PW_Latch(1,11b)   'latch both counters 1+2
    P2_Cnt_Get_PW_HL(1,1,Par_1,Par_2) 'read high/low time
    P2_Cnt_Get_PW(1,1,FPar_1,FPar_2) 'read frequency / duty
    cycle
```

## P2\_Cnt\_PW\_Latch

## P2\_Cnt\_Read

**P2\_Cnt\_Read** transfers a current counter value into Latch A and returns the value.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Cnt_Read(module, counter_no)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>counter_no</b>	Counter number: 1...4.	LONG
<b>ret_val</b>	Counter values.	LONG

### Notes

Use the return value in calculations only with variables of the type **Long** (e.g. differences or count direction).

### See also

[P2\\_Cnt\\_Clear](#), [P2\\_Cnt\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#), [P2\\_Cnt\\_Latch](#), [P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_Read4](#), [P2\\_Cnt\\_Read\\_Latch](#), [P2\\_Cnt\\_Read\\_Latch4](#)

### Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim old_1, new_1 As Long
Dim old_2, new_2 As Long

Init:
    old_1 = 0                'initialize
    old_2 = 0
    Rem Counter 1: clock-dir, enable CLR input
    P2_Cnt_Mode(1,1,10000b)
    P2_Cnt_Mode(1,2,0)      'all counters use external
trigger
    P2_Cnt_Clear(1,11b)     'reset counters 1+2 to 0
    P2_Cnt_Enable(1,11b)   'start counters 1+2, stop
counters 3+4

Event:
    P2_Cnt_Latch(1,11b)    'latch both counters 1+2
    new_1 = P2_Cnt_Read_Latch(1,1)'read latch A counter 1 and...
    new_2 = P2_Cnt_Read_Latch(1,2)'latch A counter 2
    Par_1 = new_1 - old_1 'get difference (f = impulses / time)
    Par_2 = new_2 - old_2 '-''-
    old_1 = new_1          'store new counter value as old
    old_2 = new_2          '-''-
```



**P2\_Cnt\_Read4** transfers a current counter value into Latch A and returns the value.

## Syntax

```
#Include ADwinPro_All.Inc

P2_Cnt_Read4(module, array[], index)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[]</b>	Destination array to store counter values.	ARRAY
		LONG
<b>index</b>	First element in <b>array[]</b> to be written into.	LONG

## Notes

Use the **array[]** values in calculations only with variables of the type **Long** (e.g. differences or count direction).

## See also

[P2\\_Cnt\\_Clear](#), [P2\\_Cnt\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#), [P2\\_Cnt\\_Latch](#), [P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_Read](#), [P2\\_Cnt\\_Read\\_Latch](#), [P2\\_Cnt\\_Read\\_Latch4](#)

## Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim Data_1[4] AS LONG
Dim old[4], new[4] As Long
Dim i As Long

Init:
    P2_Cnt_Enable(1,0)           'stop counters
    Rem Counter 1: clock-dir, enable CLR input
    P2_Cnt_Mode(1,1,10000b)
    P2_Cnt_Mode(1,2,0)           'all counters use external
    trigger
    Rem Counter 3: clock-dir, enable CLR input
    P2_Cnt_Mode(1,3,10000b)
    P2_Cnt_Mode(1,4,0)           'all counters use external
    trigger
    P2_Cnt_Clear(1,1111b)        'reset counters to 0
    P2_Cnt_Enable(1,1111b) 'start counters

Event:
    P2_Cnt_Read4(1,new,1)        'read counter values into array
    new
    For i = 1 To 4
        Data_1[i] = new[i]-old[i] 'get difference (f = impulses /
    time)
        old[i] = new[i]           'store new counter value as old
    Next i
```

## P2\_Cnt\_Read4

## P2\_Cnt\_Read\_Int\_Register

T11

TiCo

**P2\_Cnt\_Read\_Int\_Register** returns the content of a counter register.

### Syntax

```
#Include ADwinGoldIII.inc / GoldIITiCo.inc

ret_val = P2_Cnt_Read_Int_Register(counter_no, reg_no)
```

### Parameters

**counter\_no** Counter number: 1...4.

LONG

**reg\_no** Key number (0...15) for a counter register, see below.

LONG

**ret\_val** Content of the counter register.

LONG

reg_no	Register
0	Latch 1 for positive edges.
1	Latch 2 for positive edges.
2	Latch 3 for positive edges.
3	Latch 1 for negative edges.
4	Latch 2 for negative edges.
5	Latch 3 for negative edges.
6	Software latch for VR counter.
7	Software latch for PWM counter.
8	Shadow register for Latch 1, positive edges.
9	Shadow register for Latch 2, positive edges.
10	Shadow register for Latch 3, positive edges.
11	Shadow register for Latch 1, negative edges.
12	Shadow register for Latch 2, negative edges.
13	Shadow register for Latch 3, negative edges.
14	Shadow register for software latch, VR counter.
15	Counter status.

### Notes

- / -

### See also

[P2\\_Cnt\\_Sync\\_Latch](#)

### Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

**P2\_Cnt\_Read\_Latch** returns the value of a counter's Latch A.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Cnt_Read_Latch(module, counter_no)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>counter_no</b>	Counter number: 1...4.	LONG
<b>ret_val</b>	Contents of Latch A .	LONG

## Notes

Use the return value in calculations only with variables of the type **Long** (e.g. differences or count direction).

## See also

[P2\\_Cnt\\_Clear](#), [P2\\_Cnt\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#), [P2\\_Cnt\\_Latch](#), [P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_Read](#), [P2\\_Cnt\\_Read4](#), [P2\\_Cnt\\_Read\\_Latch4](#)

## Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim old_1, new_1 As Long
Dim old_2, new_2 As Long

Init:
    old_1 = 0                'initialize
    old_2 = 0
    Rem Counter 1: clock-dir, enable CLR input
    P2_Cnt_Mode(1,1,10000b)
    P2_Cnt_Mode(1,2,0)      'all counters use external
    trigger
    P2_Cnt_Clear(1,11b)     'reset counters 1+2 to 0
    P2_Cnt_Enable(1,11b)   'start counters 1+2, stop
    counters 3+4

Event:
    P2_Cnt_Latch(1,11b)     'latch both counters 1+2
    new_1 = P2_Cnt_Read_Latch(1,1)'read latch A counter 1 and...
    new_2 = P2_Cnt_Read_Latch(1,2)'latch A counter 2
    Par_1 = new_1 - old_1 'get difference (f = impulses / time)
    Par_2 = new_2 - old_2 '-''-
    old_1 = new_1          'store new counter value as old
    old_2 = new_2          '-''-
```

## P2\_Cnt\_Read\_Latch

## P2\_Cnt\_Read\_Latch4

**P2\_Cnt\_Read\_Latch4** returns the value of a counter's Latch A.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Cnt_Read_Latch4(module, array[], index)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[ ]</b>	Destination array to store counter values.	ARRAY
		LONG
<b>index</b>	First element in <b>array[ ]</b> to be written into.	LONG

### Notes

Use the **array[ ]** values in calculations only with variables of the type **Long** (e.g. differences or count direction).

### See also

[P2\\_Cnt\\_Clear](#), [P2\\_Cnt\\_Enable](#), [P2\\_Cnt\\_Get\\_Status](#), [P2\\_Cnt\\_Latch](#),  
[P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_Read](#), [P2\\_Cnt\\_Read4](#), [P2\\_Cnt\\_Read\\_Latch](#)

### Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim Data_1[4] AS LONG
Dim old[4], new[4] As Long
Dim i As Long

Init:
    P2_Cnt_Enable(1,0)           'stop counters
    Rem Counter 1: clock-dir, enable CLR input
    P2_Cnt_Mode(1,1,10000b)
    P2_Cnt_Mode(1,2,0)           'all counters use external
    trigger
    Rem Counter 3: clock-dir, enable CLR input
    P2_Cnt_Mode(1,3,10000b)
    P2_Cnt_Mode(1,4,0)           'all counters use external
    trigger
    P2_Cnt_Clear(1,1111b)        'reset counters to 0
    P2_Cnt_Enable(1,1111b) 'start counters

Event:
    P2_Cnt_Latch(1,1111b)        'latch counters
    P2_Cnt_Read_Latch4(1,new,1) 'read counter values into array
    new
    For i = 1 To 4
        Data_1[i] = new[i]-old[i] 'get difference (f = impulses /
    time)
        old[i] = new[i]           'store new counter value as old
    Next i
```

**P2\_Cnt\_Sync\_Latch** copies the contents of selected counters and PWM counters into latches.

## Syntax

```
#Include ADwinPro_All.inc

P2_Cnt_Sync_Latch(module, pattern)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern to select pairs of counters. Bit = 0: No counter selected. Bit = 1: Copy counter content into latch.	LONG

Bit no.	31...4	3	2	1	0
Counter no.	–	4	3	2	1

## Notes

Each bit is associated either to a standard counter and a PWM counter. Both counter contents are latched at the same time. Therefore, the instruction has the same function as both **P2\_Cnt\_Latch** and **P2\_Cnt\_PW\_Latch**.

The latches can be read e.g. with **P2\_Cnt\_Read\_Latch** or **P2\_Cnt\_Get\_PW**.

## See also

[P2\\_Cnt\\_Get\\_PW](#), [P2\\_Cnt\\_Latch](#), [P2\\_Cnt\\_Mode](#), [P2\\_Cnt\\_PW\\_Latch](#), [P2\\_Cnt\\_Read\\_Latch](#), [P2\\_Sync\\_All](#)

## Valid for

CNT-D Rev. E, CNT-I Rev. E, CNT-T Rev. E, MIO-4-ET1 Rev. E

## P2\_Cnt\_Sync\_Latch

### Example

```
#Include ADwinPro_All.inc
#Define frequency FPar_1
Dim time, edges As Long
Dim rest As Float
Dim oldpw, oldcnt, newpw, newcnt As Long
Dim pw_cnt As Long

Init:
    Processdelay = 3000000      '100Hz
    P2_Cnt_Enable(1,0001b)
    P2_Cnt_Mode(1,1,00000000b) 'mode: clock/dir
    P2_Cnt_Clear(1,0Fh)
    P2_Cnt_Enable(1,0Fh)      'enable standard counters
    P2_Cnt_PW_Enable(1,0Fh)   'enable PW counters
    P2_Cnt_PW_Latch(1,0Fh)    'copy all counter values
    oldpw = 0
    oldcnt = 0
    frequency = 0

Event:
    P2_Cnt_Sync_Latch(1,0001b) 'latch all counter values
    newcnt = P2_Cnt_Read_Latch(1,1) 'vr-cnt
    edges = (newcnt-oldcnt)      'number of edges between events
    If (edges <> 0) Then
        PW_cnt = P2_Cnt_Read_Int_Register(1,1,8)
        time = PW_cnt - oldpw    'calculate timebase
        frequency = edges*100000000/time 'frequency
                                     '(100000000->frequency of
P2-CNT-Module)
        oldcnt=newcnt           'store VR-counter value
        oldpw =newpw           'store PW-counter value
    EndIf
```

**P2\_SSI\_Mode** sets the modes of all SSI decoders on the specified module, either "single shot" (read out once) or "continuous" (read out continuously).

## Syntax

```
#Include ADwinPro_All.Inc

P2_SSI_Mode(module,pattern)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Operation mode of the SSI decoders, indicated as bit pattern. A bit is assigned to each of the decoders (see table). Bit = 0: "Single shot" mode, the encoder is read out once. Bit = 1: "Continuous" mode, the encoder is read out continuously.	LONG

Bit no.	31:2	1	0
SSI decoder	–	2	1

## Notes

If you select the mode "continuous", reading the encoder is started immediately. **P2\_SSI\_Start** is not necessary for this. With **P2\_SSI\_Set\_Delay**, you set the time delay between reading two encoder values.

Using the "continuous" mode, some encoder types occasionally return the wrong counter value 0 (zero) instead of the correct counter value. This error does not occur with the "single shot" mode.

## See also

[P2\\_SSI\\_Read](#), [P2\\_SSI\\_Read2](#), [P2\\_SSI\\_Set\\_Bits](#), [P2\\_SSI\\_Set\\_Clock](#), [P2\\_SSI\\_Set\\_Delay](#), [P2\\_SSI\\_Start](#), [P2\\_SSI\\_Status](#)

## Valid for

CNT-D Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Init:
P2_SSI_Set_Clock(1,200) 'CLK (clock rate) = 250 kHz
P2_SSI_Set_Delay(1,1,250)'Waiting delay decoder 1: 5µs
P2_SSI_Set_Delay(1,2,500)'Waiting delay decoder 2: 10µs
P2_SSI_Set_Bits(1,1,23) 'Amount of bits=23 (decoder 1)
P2_SSI_Set_Bits(1,2,23) 'Amount of bits=23 (decoder 2)
P2_SSI_Mode(1,3) 'continuous-mode for both decoders

Event:
Par_1 = P2_SSI_Read(1,1) 'Read position value decoder 1
Par_2 = P2_SSI_Read(1,2) 'Read position value decoder 2
```

## P2\_SSI\_Mode

## P2\_SSI\_Read

**P2\_SSI\_Read** returns the last saved counter value of a specified SSI counter on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
ret_val = P2_SSI_Read(module, dcd_r_no)
```

### Parameters

<code>module</code>	Specified module address (1...15).	LONG
<code>dcd_r_no</code>	Number (1, 2) of the SSI decoder whose counter value is to be read.	LONG
<code>ret_val</code>	Previous counter value of the SSI counter (= absolute value position of the encoder).	LONG

### Notes

An encoder value is saved when the bits indicated by **P2\_SSI\_Set\_Bits** are read.



Always the amount of bits is returned that is set before by **P2\_SSI\_Set\_Bits**, even if this does not correspond to the resolution of the encoder.

In this case the returned counter value depends on the encoder (see documentation of the manufacturer). Normally there are the following rules:

- If the encoder has a higher resolution, its exceeding least-significant bits are not used.
- If the encoder has a lower resolution as indicated, a 0 (zero) is read for each missing most-significant bit.

### See also

[P2\\_SSI\\_Mode](#), [P2\\_SSI\\_Read2](#), [P2\\_SSI\\_Set\\_Bits](#), [P2\\_SSI\\_Set\\_Clock](#), [P2\\_SSI\\_Set\\_Delay](#), [P2\\_SSI\\_Start](#), [P2\\_SSI\\_Status](#)

### Valid for

CNT-D Rev. E, MIO-4-ET1 Rev. E



## Example

```
#Include ADwinPro_All.Inc
Dim m, n, y As Long

Init:
  P2_SSI_Set_Clock(1,50)   'CLK (clock rate) = 1 MHz
  P2_SSI_Set_Delay(1,1,250)'Waiting delay decoder: 5µs
  P2_SSI_Set_Bits(1,1,23)  'Amount of bits=23 (decoder 1)
  P2_SSI_Mode(1,1)         'Set continuous-mode (decoder 1)

Event:
  Par_1 = P2_SSI_Read(1,1) 'Read out and display position
                           'value (decoder 1)
  REM If you have an encoder with Gray-code:
  m = 0                    'delete value of the last conversion
  y = 0                    ' -"-
  For n = 1 To 32          'Check all 32 possible bits
    m = (Shift_Right(Par_1,(32 - n)) And 1) XOr m
    y = (Shift_Left(m,(32 - n))) Or y
  Next n
  Par_9 = y                'The result of the Gray/binary
                           'conversion in Par_9
```

## P2\_SSI\_Read2

**P2\_SSI\_Read2** returns the last saved counter values of both SSI counters on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_SSI_Read2(module,array[],index)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>array[ ]</b>	Destination array to store counter values.	ARRAY
		LONG
<b>index</b>	First element in <b>array[ ]</b> to be written into.	LONG

### Notes

An encoder value is saved when the bits indicated by **P2\_SSI\_Set\_Bits** are read.



Always the amount of bits is returned that is set before by **P2\_SSI\_Set\_Bits**, even if this does not correspond to the resolution of the encoder.

In this case the returned counter value depends on the encoder (see documentation of the manufacturer). Normally there are the following rules:

- If the encoder has a higher resolution, its exceeding least-significant bits are not used.
- If the encoder has a lower resolution as indicated, a 0 (zero) is read for each missing most-significant bit.

### See also

[P2\\_SSI\\_Mode](#), [P2\\_SSI\\_Read](#), [P2\\_SSI\\_Set\\_Bits](#), [P2\\_SSI\\_Set\\_Clock](#), [P2\\_SSI\\_Set\\_Delay](#), [P2\\_SSI\\_Start](#), [P2\\_SSI\\_Status](#)

### Valid for

CNT-D Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim Data_1[2000] As Long

Init:
P2_SSI_Set_Clock(1,50) 'CLK (clock rate) = 1 MHz
P2_SSI_Set_Delay(1,1,250) 'Waiting delay decoder 1: 5µs
P2_SSI_Set_Delay(1,2,500) 'Waiting delay decoder 2: 10µs
P2_SSI_Set_Bits(1,1,10) '10 bits for decoder 1
P2_SSI_Set_Bits(1,2,25) '25 bits for decoder 2
P2_SSI_Mode(1,3) 'Set continuous-mode (both decoders)
Par_1 = 0

Event:
Inc Par_1
If (Par_1 > 1000) Then Par_1 = 1
P2_SSI_Read2(1,Data_1,Par_1*2) 'Read both position values
```

**P2\_SSI\_Set\_Bits** sets for an SSI counter on the specified module the amount of bits which generate a complete encoder value.

The number of bits should be similar to the resolution of the encoder.

## Syntax

```
#Include ADwinPro_All.Inc

P2_SSI_Set_Bits(module, dcd_r_no, bit_no)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>dcd_r_no</b>	Number (1, 2) of the SSI decoder whose resolution is to be set.	LONG
<b>bit_no</b>	Amount of bits (1...32) to be read for the encoder value (corresponds to encoder resolution).	LONG

## Notes

The resolution (amount of bits) of the SSI encoder should be similar to the amount of transferred bits.

It is always expected to get that certain amount of bits for an encoder value that was indicated before by **P2\_SSI\_Set\_Bits**, even if this does not correspond to the resolution of the encoder.

In this case the returned counter value depends on the encoder (see documentation of the manufacturer). Normally there are the following rules:

- If the encoder has a higher resolution, its exceeding least-significant bits are not used.
- If the encoder has a lower resolution as indicated, a 0 (zero) is read for each missing most-significant bit.

## See also

[P2\\_SSI\\_Mode](#), [P2\\_SSI\\_Read](#), [P2\\_SSI\\_Read2](#), [P2\\_SSI\\_Set\\_Clock](#), [P2\\_SSI\\_Set\\_Delay](#), [P2\\_SSI\\_Start](#), [P2\\_SSI\\_Status](#)

## Valid for

CNT-D Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Init:
P2_SSI_Set_Clock(1,50) 'CLK (clock rate) = 1 MHz
P2_SSI_Set_Delay(1,1,250)'Waiting delay decoder 1: 5µs
P2_SSI_Set_Delay(1,2,500)'Waiting delay decoder 2: 10µs
P2_SSI_Mode(1,3) 'Set continuous-mode (both decoders)
P2_SSI_Set_Bits(1,1,10) '10 bits for decoder 1
P2_SSI_Set_Bits(1,2,25) '25 bits for decoder 2

Event:
Par_1 = P2_SSI_Read(1,1) 'Read position value decoder 1
Par_2 = P2_SSI_Read(1,2) 'Read position value decoder 2
```

## P2\_SSI\_Set\_Bits



## P2\_SSI\_Set\_Clock

**P2\_SSI\_Set\_Clock** sets the clock rate (approx. 100kHz to 2.5MHz) on the specified module, with which the decoder is clocked.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_SSI_Set_Clock(module,prescale)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>prescale</b>	scale factor (1...4095) for setting the clock rate according to the equation: Clock rate = 50MHz / <b>prescale</b> .	LONG

### Notes



The setting of the clock rate is always identical for both encoders, which are connected to the module, and cannot be set separately. If necessary, the clock has to consider the clock rate of the slowest encoder.

After start-up of the module the default scale factor of 100 is used, corresponding to 500kHz.

For scale factors > 255 only the least significant 12 bits are used as scale factor.

The possible clock frequency depends on the length of the cable, cable type, and the send and receive components of the encoder or decoder. Basically the following rule applies: The higher the clock frequency the shorter the cable length.

### See also

[P2\\_SSI\\_Mode](#), [P2\\_SSI\\_Read](#), [P2\\_SSI\\_Read2](#), [P2\\_SSI\\_Set\\_Bits](#), [P2\\_SSI\\_Set\\_Delay](#), [P2\\_SSI\\_Start](#), [P2\\_SSI\\_Status](#)

### Valid for

CNT-D Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
  
Init:  
P2_SSI_Set_Clock(1,10) 'CLK (clock rate) = 5 MHz  
P2_SSI_Set_Delay(1,1,250) 'Waiting delay decoder 1: 5µs  
P2_SSI_Set_Delay(1,2,500) 'Waiting delay decoder 2: 10µs  
P2_SSI_Mode(1,3) 'Set continuous-mode  
' (for both decoders)  
P2_SSI_Set_Bits(1,1,10) 'Amount of bits=10 (decoder 1)  
P2_SSI_Set_Bits(1,2,25) 'Amount of bits=25 (decoder 2)  
  
Event:  
Par_1 = P2_SSI_Read(1,1) 'Read position value decoder 1  
Par_2 = P2_SSI_Read(1,2) 'Read position value decoder 2
```

**P2\_SSI\_Set\_Delay** sets the waiting time between reading two encoder values for one SSI-decoder on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

P2_SSI_Set_Delay(module, dcd_r_no, delay)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>dcd_r_no</b>	Number (1, 2) of the SSI decoder whose waiting time is to be set.	LONG
<b>delay</b>	Waiting time (1...65535) in units of 20ns; the selectable range is 20ns...1310.7µs	LONG

## Notes

The waiting time **delay** starts after an encoder value is read completely and ends when the next encoder value starts being read.

After start-up of the module the default value of 1250 is used, corresponding to 25µs.

## See also

[P2\\_SSI\\_Mode](#), [P2\\_SSI\\_Read](#), [P2\\_SSI\\_Read2](#), [P2\\_SSI\\_Set\\_Bits](#), [P2\\_SSI\\_Set\\_Clock](#), [P2\\_SSI\\_Start](#), [P2\\_SSI\\_Status](#)

## Valid for

CNT-D Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Init:
P2_SSI_Set_Clock(1,50)    'CLK (clock rate) = 1 MHz
P2_SSI_Set_Delay(1,1,400)'waiting time 8µs for decoder 1
P2_SSI_Set_Delay(1,2,200)'waiting time 4µs for decoder 2
P2_SSI_Set_Bits(1,1,10)  '10 bits for decoder 1
P2_SSI_Set_Bits(1,2,25)  '25 bits for decoder 2
P2_SSI_Mode(1,3)         'Set continuous-mode (for both
                           'decoders)

Event:
Par_1 = P2_SSI_Read(1,1) 'Read position value decoder 1
Par_2 = P2_SSI_Read(1,2) 'Read position value decoder 2
```

## P2\_SSI\_Set\_De- lay

## P2\_SSI\_Start

**P2\_SSI\_Start** starts the reading of one or both SSI decoders on the specified module (only in mode "single shot").

### Syntax

```
#Include ADwinPro_All.Inc

P2_SSI_Start(module,pattern)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern for selecting the SSI decoders which are to be started: Bit = 0: No function. Bit = 1: Start reading of the SSI decoder.	LONG

Bit no.	31:2	1	0
SSI decoder	–	2	1

### Notes

In the continuous mode this instruction has no function, because the encoder values are nevertheless read out continuously.

An encoder value will be saved only when the amount of bits is read which is set by **P2\_SSI\_Set\_Bits**.

A complete encoder value is always transferred, even if the operation mode is changing meanwhile.

### See also

[P2\\_SSI\\_Mode](#), [P2\\_SSI\\_Read](#), [P2\\_SSI\\_Read2](#), [P2\\_SSI\\_Set\\_Bits](#), [P2\\_SSI\\_Set\\_Clock](#), [P2\\_SSI\\_Set\\_Delay](#), [P2\\_SSI\\_Status](#)

### Valid for

CNT-D Rev. E, MIO-4-ET1 Rev. E

### Example

```
#Include ADwinPro_All.Inc

Init:
P2_SSI_Set_Clock(1,250) 'CLK (clock rate) = 200 kHz
P2_SSI_Set_Delay(1,1,250)'Waiting delay decoder 1: 5µs
P2_SSI_Set_Delay(1,2,500)'Waiting delay decoder 2: 10µs
P2_SSI_Mode(1,0) 'Set single shot-mode
' (both counters)

P2_SSI_Set_Bits(1,1,23) 'Amount of bits=23 (decoder 1)
P2_SSI_Set_Bits(1,2,23) 'Amount of bits=23 (decoder 2)

Event:
P2_SSI_Start(1,3) 'Read position value of decoders 1 & 2
Do 'for decoder 1:
Until (P2_SSI_Status(1,1) = 0)
REM If position value is read completely, then ...
Par_1 = P2_SSI_Read(1,1) 'read out and display position value
Do 'For decoder 2:
Until (P2_SSI_Status(1,2) = 0)
REM If position value is read completely, then ...
Par_1 = P2_SSI_Read(1,2) 'read out and display position value
```



**P2\_SSI\_Status** returns the current read-status on the specified module for a specified decoder.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_SSI_Status(module, dcd_r_no)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>dcd_r_no</b>	Number (1, 2) of the SSI decoder whose status is to be queried.	LONG
<b>ret_val</b>	Read-status of the decoder: 0: Decoder is ready, that is a complete value has been read. 1: Decoder is reading an encoder value.	LONG

## Notes

Use the status query only in the SSI mode "single shot". In the mode "continuous" querying the status is not useful.

## See also

[P2\\_SSI\\_Mode](#), [P2\\_SSI\\_Read](#), [P2\\_SSI\\_Read2](#), [P2\\_SSI\\_Set\\_Bits](#), [P2\\_SSI\\_Set\\_Clock](#), [P2\\_SSI\\_Set\\_Delay](#), [P2\\_SSI\\_Start](#)

## Valid for

CNT-D Rev. E, MIO-4-ET1 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Init:
P2_SSI_Set_Clock(1,250) 'CLK (clock rate) = 200 kHz
P2_SSI_Mode(1,0)        'Set single shot-mode
                        '(both counters)
P2_SSI_Set_Bits(1,1,23) 'Amount of bits=23 (decoder 1)
P2_SSI_Set_Bits(1,2,23) 'Amount of bits=23 (decoder 2)

Event:
P2_SSI_Start(1,3)        'Read position value of decoders 1 & 2
Do                        'For decoder 1:
Until (P2_SSI_Status(1,1) = 0)
REM If position value is read completely, then ...
Par_1 = P2_SSI_Read(1,1) 'Read out and display position value
Do                        'For decoder 2:
Until (P2_SSI_Status(1,2) = 0)
REM If position value is read completely, then ...
Par_1 = P2_SSI_Read(1,2) 'Read out and display position value
```

## P2\_SSI\_Status

### 3.7 Pro II: PWM Output Modules

This section describes instructions which apply to Pro II PWM output modules:

- [P2\\_PWM\\_Enable](#) (page 189)
- [P2\\_PWM\\_Get\\_Status](#) (page 190)
- [P2\\_PWM\\_Init](#) (page 191)
- [P2\\_PWM\\_Latch](#) (page 193)
- [P2\\_PWM\\_Reset](#) (page 194)
- [P2\\_PWM\\_Standby\\_Value](#) (page 195)
- [P2\\_PWM\\_Write\\_Latch](#) (page 196)
- [P2\\_PWM\\_Write\\_Latch\\_Block](#) (page 197)

In the [Instruction List sorted by Module Types](#) (annex A.2) you will find which of the functions corresponds to the *ADwin-Pro II* modules.

It is presumed that application examples use the module address 1.





**P2\_PWM\_Enable** enables or disables one or more PWM outputs.

## Syntax

```
#Include ADwinPro_All.Inc

P2_PWM_Enable(module, pattern)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern for selection of PWM outputs. Bit = 0: Disable PWM output. Bit = 1: Enable PWM output.	LONG

Bit no.	31...1	15	14	...	1	0
	6					
PWM output	–	16	15	...	2	1

## Notes

The time, when the PWM outputs are disabled—at once or after the next end of period—depends on the setting which was done with **P2\_PWM\_Init** (parameter **mode**).

## See also

[P2\\_PWM\\_Get\\_Status](#), [P2\\_PWM\\_Init](#), [P2\\_PWM\\_Latch](#), [P2\\_PWM\\_Reset](#), [P2\\_PWM\\_Standby\\_Value](#), [P2\\_PWM\\_Write\\_Latch](#)

## Valid for

PWM-16(-I) Rev. E

## Example

see [P2\\_PWM\\_Init](#) (page 191)

## P2\_PWM\_Enable

## P2\_PWM\_Get\_Status

**P2\_PWM\_Get\_Status** returns the operation status of all PWM outputs.

### Syntax

```
#Include ADwinPro_All.Inc  
  
ret_val = P2_PWM_Get_Status(module)
```

### Parameters

module	Specified module address (1...15).	LONG
ret_val	Bit pattern with status bits of all PWM outputs. Bit = 0: PWM output is disabled. Bit = 1: PWM output is enabled.	LONG

Bit no.	31...1 6	15	14	...	1	0
PWM output	–	16	15	...	2	1

### Notes

- / -

### See also

[P2\\_PWM\\_Enable](#), [P2\\_PWM\\_Init](#), [P2\\_PWM\\_Latch](#), [P2\\_PWM\\_Reset](#),  
[P2\\_PWM\\_Standby\\_Value](#), [P2\\_PWM\\_Write\\_Latch](#)

### Valid for

PWM-16(-I) Rev. E

### Example

- / -

**P2\_PWM\_Init** sets the defaults for one PWM output.

## Syntax

```
#Include ADwinPro_All.Inc

P2_PWM_Init(module, pwm_output, startdelay,
            startvalue,
            mode, count)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pwm_output</b>	Number (1...16) of PWM output.	LONG
<b>startdelay</b>	Start delay in units of 10ns.	LONG
<b>startvalue</b>	Start level of PWM output: 0: TTL-level low. 1: TTL-level high.	LONG
<b>mode</b>	Operating mode of PWM output as bit pattern (bits 0...2 only). Bit 0: Moment to take over a new PW frequency: <ul style="list-style-type: none"> <li>Bit = 0: Take over at end of period.</li> <li>Bit = 1: Take over immediately.</li> </ul> Bit 1: Number of pulses: <ul style="list-style-type: none"> <li>Bit = 0: infinite number of pulses.</li> <li>Bit = 1: number of pulses is <b>count</b>.</li> </ul> Bit 2: Moment to stop after stop instruction: <ul style="list-style-type: none"> <li>Bit = 0: Stop at end of period.</li> <li>Bit = 1: Stop immediately.</li> </ul>	LONG
<b>count</b>	Number of periods (1...32768), which are processed during an output cycle.	LONG

## Notes

- / -

## See also

[P2\\_PWM\\_Enable](#), [P2\\_PWM\\_Get\\_Status](#), [P2\\_PWM\\_Latch](#), [P2\\_PWM\\_Reset](#), [P2\\_PWM\\_Standby\\_Value](#), [P2\\_PWM\\_Write\\_Latch](#)

## Valid for

PWM-16(-I) Rev. E

## P2\_PWM\_Init

### Example

```
#Include ADwinPro_All.inc
#Define module 4
#Define freq1 FPar_1
#Define freq2 FPar_2
#Define pw1 FPar_3
#Define pw2 FPar_4
Dim channel As Long

Init:
    freq1 = 1000           '1000 Hz
    freq2 = 2000           '2000 Hz
    pw1 = 50               '50 %
    pw2 = 70               '70 %
    P2_PWM_Reset(module,011B) 'stop channels 1 und 2

    For channel = 1 To 2
        P2_PWM_Init(module,channel,0,0,0,0)
    Next

    P2_PWM_Write_Latch(module,1,pw1,freq1)
    P2_PWM_Write_Latch(module,2,pw2,freq2)
    P2_PWM_Latch(module,11B)
    P2_PWM_Enable(module,011B) 'start output

Event:
    P2_PWM_Write_Latch(module,1,pw1,freq1)
    P2_PWM_Write_Latch(module,2,pw2,freq2)

    P2_PWM_Latch(module,11B)
```

**P2\_PWM\_Latch** enables frequency and duty cycle of one or more PWM outputs to be output.

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_PWM_Latch(module, pattern)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern to select PWM outputs: Bit = 0: No influence. Bit = 1: latch = enable for output.	LONG

Bit no.	31...1 6	15	14	...	1	0
PWM output	–	16	15	...	2	1

## Notes

**P2\_PWM\_Write\_Latch** writes frequency and duty cycle into the latch register. Only when **P2\_PWM\_Latch** is processed the latch values are started to be output.

The time, when the output of the new values starts—at once or after the next end of period—depends on the setting which was done with **P2\_PWM\_Init** (parameter **mode**).

## See also

[P2\\_PWM\\_Enable](#), [P2\\_PWM\\_Get\\_Status](#), [P2\\_PWM\\_Init](#), [P2\\_PWM\\_Reset](#), [P2\\_PWM\\_Standby\\_Value](#), [P2\\_PWM\\_Write\\_Latch](#)

## Valid for

PWM-16(-I) Rev. E

## Example

see [P2\\_PWM\\_Init](#) (page 191)

## P2\_PWM\_Latch

## P2\_PWM\_Reset

**P2\_PWM\_Reset** stops the output of one or more PWM outputs immediately..

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_PWM_Reset(module,pattern)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pattern</b>	Bit pattern to select the PWM outputs: Bit = 0: No influence Bit = 1: Stop output immediately.	LONG

Bit no.	31...1 6	15	14	...	1	0
PWM output	–	16	15	...	2	1

### Notes

The output will be stopped immediately even when **P2\_PWM\_Init** has set a different stop mode.

### See also

[P2\\_PWM\\_Enable](#), [P2\\_PWM\\_Get\\_Status](#), [P2\\_PWM\\_Init](#), [P2\\_PWM\\_Latch](#), [P2\\_PWM\\_Standby\\_Value](#), [P2\\_PWM\\_Write\\_Latch](#)

### Valid for

PWM-16(-I) Rev. E

### Example

see [P2\\_PWM\\_Init](#) (page 191)

**P2\_PWM\_Standby\_Value** sets the default TTL levels for all PWM outputs.

## Syntax

```
#Include ADwinPro_All.Inc

P2_PWM_Standby_Value(module,pattern)
```

## Parameters

**module** Specified module address (1...15). LONG

**pattern** Bit pattern to select the default TTL level of the PWM outputs: LONG

Bit = 0: TTL-level low  
Bit = 1: TTL-level high

Bit no.	31...1 6	15	14	...	1	0
PWM output	–	16	15	...	2	1

## Notes

Using **P2\_PWM\_Standby\_Value**, PWM outputs may be used as simple TTL outputs.

If the PWM output is disabled with **P2\_PWM\_Enable**, the output is set to default level from **pattern**. The default level will also be set after the PWM output has stopped.

After power-up the outputs are set to TTL-level low.

## See also

[P2\\_PWM\\_Enable](#), [P2\\_PWM\\_Get\\_Status](#), [P2\\_PWM\\_Init](#), [P2\\_PWM\\_Latch](#), [P2\\_PWM\\_Reset](#), [P2\\_PWM\\_Write\\_Latch](#)

## Valid for

PWM-16(-I) Rev. E

## Example

- / -

## P2\_PWM\_Standby\_Value

## P2\_PWM\_Write\_Latch

**P2\_PWM\_Write\_Latch** writes frequency and duty cycle into the latch register.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_PWM_Write_Latch(module, pwm_output, dutycycle, frequency)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>pwm_output</b>	Number (1...16) of PWM output.	LONG
<b>dutycycle</b>	Duty cycle / inverse duty cycle in percent between 0.0 and 100.0 (do not use 0.0 or 100.0).	FLOAT
<b>frequency</b>	Frequency in Hertz: 0.025Hz ...500kHz.	FLOAT

### Notes

The value of **dutycycle** depends on the setting of the parameter **startvalue** from the instruction **PWM\_Init**:

- **startvalue** = 1: Set **dutycycle** to the value of the duty cycle.
- **startvalue** = 0: Set **dutycycle** to the "inverse duty cycle":  
 $\text{dutycycle} = 100\% - \text{duty cycle}$

The highest output frequency where the duty cycle can be still defined in 1%-steps, is 500kHz.

### See also

[P2\\_PWM\\_Enable](#), [P2\\_PWM\\_Get\\_Status](#), [P2\\_PWM\\_Init](#), [P2\\_PWM\\_Latch](#), [P2\\_PWM\\_Reset](#), [P2\\_PWM\\_Standby\\_Value](#)

### Valid for

PWM-16(-I) Rev. E

### Example

see [P2\\_PWM\\_Init](#) (page 191)



**P2\_PWM\_Write\_Latch\_Block** writes frequency and duty cycle for several PWM outputs into the latch registers.

## Syntax

```
#Include ADwinPRO_ALL.inc

P2_PWM_Write_Latch_Block(module, dutycycle[],
    frequency[], channel_count)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>dutycycle</b> [ ]	Duty cycle / inverse duty cycle in percent between 0.0 and 100.0 (do not use 0.0 or 100.0).	FLOAT
<b>frequency</b> [ ]	Frequency in Hertz: 0.025Hz ...100MHz.	FLOAT
<b>channel_ count</b>	Number (1...16) of output channels, where output data is to be set.	LONG

## Notes

The value of **dutycycle** depends on the setting of the parameter **startvalue** from the instruction **PWM\_Init**:

- **startvalue** = 1: Set **dutycycle** to the value of the duty cycle.
- **startvalue** = 0: Set **dutycycle** to the "inverse duty cycle":  
**dutycycle** = 100% - duty cycle

The output data are set for the PWM outputs 1...**channel\_count**.

Using **P2\_PWM\_Write\_Latch\_Block**, frequency and duty cycle are only written into the latch registers. **P2\_PWM\_Latch** has to be used to activate the values for PWM output.

The highest output frequency where the duty cycle can be still defined in 1%-steps, is 1 MHz.

## See also

[P2\\_PWM\\_Enable](#), [P2\\_PWM\\_Get\\_Status](#), [P2\\_PWM\\_Init](#), [P2\\_PWM\\_Latch](#), [P2\\_PWM\\_Reset](#), [P2\\_PWM\\_Standby\\_Value](#)

## Valid for

PWM-16(-I) Rev. E

## P2\_PWM\_Write\_Latch\_Block

#### Example

```
#Include ADwinPro_All.inc
#Define module 4
#Define freq Data_1
#Define pw Data_2
Dim freq[16] As Float
Dim pw[16] As Float
Dim channel As Long

Init:
  For channel = 1 To 16
    freq[channel] = 1000 * channel 'channel 1: 1 kHz, channel
16: 16 KHz
    pw[channel] = 50 'all channels 50 %
  Next
  P2_PWM_Reset(module,0FFFFh)'stop all channels
  For channel = 1 To 16
    P2_PWM_Init(module,channel,0,0,0,0)
  Next
  P2_PWM_Write_Latch_Block(module, pw, freq, 3)
  P2_PWM_Latch(module,0FFFFh)
  P2_PWM_Enable(module,0FFFFh)'start output

Event:
  P2_PWM_Write_Latch_Block(module, pw, freq, 3)
  P2_PWM_Latch(module,11b)
```

### 3.8 Pro II: Temperature Measuring Modules

This section describes instructions for Pro II modules for temperature measurement:

- [P2\\_RTD\\_Channel\\_Config](#) (page 200)
- [P2\\_RTD\\_Config](#) (page 202)
- [P2\\_RTD\\_Convert](#) (page 203)
- [P2\\_RTD\\_Read](#) (page 204)
- [P2\\_RTD\\_Read8](#) (page 205)
- [P2\\_RTD\\_Start](#) (page 206)
- [P2\\_RTD\\_Status](#) (page 208)
- [P2\\_TC\\_Latch](#) (page 209)
- [P2\\_TC\\_Read\\_Latch](#) (page 210)
- [P2\\_TC\\_Read\\_Latch4](#) (page 212)
- [P2\\_TC\\_Read\\_Latch8](#) (page 214)
- [P2\\_TC\\_Set\\_Rate](#) (page 216)

In the [Instruction List sorted by Module Types](#) (annex A.2) you will find which of the functions corresponds to the *ADwin-Pro II* modules.

It is presumed that application examples use the module address 1.



## P2\_RTD\_Channel\_Config

**P2\_RTD\_Channel\_Config** sets the temperature measuring mode for a certain channel on the specified module.

### Syntax

```
#Include ADwinPro_All.inc
```

```
P2_RTD_Channel_Config(module, channel, active, type,  
element, filter, sample_period)
```

### Parameters

<code>module</code>	Specified module address (1...15).	LONG
<code>channel</code>	Number (1...8) of the measuring channel.	LONG
<code>active</code>	Activation of the measuring channel: 0: Channel is disabled. 1: Channel is enabled.	LONG
<code>type</code>	Measuring method (0...2): 0: 2 wire 1: 4 wire 2: 3 wire	LONG
<code>element</code>	Type (0...2) of thermo couple: 0: PT100 1: PT500 2: PT1000 3: Ni100	LONG
<code>filter</code>	Filter quality (0...7); in order to filter periodic disturbances, a number of measurements is averaged to a measuring value: 0: 1 measurement (filter off). 1: 2 measurements ( $= 2^1$ ). 2: 4 measurements ( $= 2^2$ ). 3: 8 measurements ( $= 2^3$ ). 4: 16 measurements ( $= 2^4$ ). 5: 32 measurements ( $= 2^5$ ). 6: 64 measurements ( $= 2^6$ ). 7: 128 measurements ( $= 2^7$ ).	LONG
<code>sample_period</code>	Sampling interval in microseconds (3...65535) between 2 measurements (not between 2 measurement values).	LONG

### Notes

After power-up of the hardware all measurement channels are disabled. Enabled measuring channels are processed with ascending channel number in the measuring cycle.

You may only enable those channels, which are connected to a temperature sensor. Otherwise there may be disturbances on the other channels—being connected to temperature sensors—which falsify the measurement values.

A measuring cycle comprises all enabled measuring channels. With "continuous" mode you can even enable or disable measuring channels during a measuring cycle.

The measuring methods are described in the hardware manual.

A high filter quality `filter` will raise the accuracy of the measurement value, but also the duration to provide a measurement value.

Using the fitting value for the sampling interval `sample_period` you

can optimize the filter for a specific disturbance frequency. Calculate the sampling interval (in microseconds) as follows:

$$\text{sample\_period} = 10^6 / (\text{frequency} \cdot 2^{\text{filter}})$$

By setting the sampling interval all measurements are equidistantly arranged along the period duration of the disturbance frequency, which therefore will be filtered from the measurement value.

The duration T for one measurement value (of a 2 or 4 wire measurement) is  $T = \text{sample\_period} \times 2^{\text{filter}}$ .

The duration T of a 3 wire measurement is double as long, since it requires double the number of measurements.

### See also

[P2\\_RTD\\_Config](#), [P2\\_RTD\\_Convert](#), [P2\\_RTD\\_Read](#), [P2\\_RTD\\_Read8](#), [P2\\_RTD\\_Start](#), [P2\\_RTD\\_Status](#)

### Valid for

RTD-8 Rev. E

### Example

see [P2\\_RTD\\_Start](#)

## P2\_RTD\_Config

**P2\_RTD\_Config** initializes the temperature measurement on the specified module.

### Syntax

```
#Include ADwinPro_All.inc  
  
P2_RTD_Config(module, mode, muxtime)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>mode</b>	Operating mode of temperature measurement: 0: Mode "single shot", single measurement cycle. 1: Mode "continuous", continuous measurement cycle.	LONG
<b>muxtime</b>	Settling time after switching to a different measuring channel: 0: Standard setting (5000 = 100µs). 125...2 <sup>31</sup> : Time in units of 20ns.	LONG

### Notes

You configure the operating mode of each temperature channel separately using **P2\_RTD\_Config\_Channel**.

A measuring cycle comprises all enabled measuring channels. With "continuous" mode you can even enable or disable measuring channels during a measuring cycle.

The longer the measuring cable to the temperature sensor the higher you should set the settling time.

The duration  $T_{\text{total}}$  of a measurement cycle is the sum of the measuring duration of the enabled temperature channels and the settling times:

$$T_{\text{total}} = \sum_1^{\text{channels}} (T_{\text{channel}} + \text{settling time})$$

### See also

[P2\\_RTD\\_Channel\\_Config](#), [P2\\_RTD\\_Convert](#), [P2\\_RTD\\_Read](#), [P2\\_RTD\\_Read8](#), [P2\\_RTD\\_Start](#), [P2\\_RTD\\_Status](#)

### Valid for

RTD-8 Rev. E

### Example

see [P2\\_RTD\\_Start](#)

**P2\_RTD\_Convert** calculates the resistance or the temperature in degrees Celsius or Fahrenheit from the measured digital value of a temperature sensor.

## Syntax

```
#Include ADwinPro_All.inc
```

```
ret_val = P2_RTD_Convert(dig_val, element, ret_type)
```

## Parameters

<b>dig_val</b>	Digital value (24 bit).	LONG
<b>element</b>	Type (0...2) of thermo couple: 0: PT100 1: PT500 2: PT1000 3: Ni100	LONG
<b>ret_type</b>	Type of return value: 0: Resistance in Ohm. 1: Temperature in degrees Celsius. 2: Temperature in degrees Fahrenheit.	LONG
<b>ret_val</b>	Resistance in Ohm or temperature in degrees Celsius or in degrees Fahrenheit.	FLOAT

## Notes

The temperature values in °C and °F apply only for standard sensors according to the norms IEC 751 (Pt) and IEC 43760 (Ni).

## See also

[P2\\_RTD\\_Channel\\_Config](#), [P2\\_RTD\\_Config](#), [P2\\_RTD\\_Read](#), [P2\\_RTD\\_Read8](#), [P2\\_RTD\\_Start](#), [P2\\_RTD\\_Status](#)

## Valid for

RTD-8 Rev. E

## Example

see [P2\\_RTD\\_Start](#)

## P2\_RTD\_Convert

## P2\_RTD\_Read

**P2\_RTD\_Read** returns the current digital measurement value of a temperature sensor at the specified channel on the module.

### Syntax

```
#Include ADwinPro_All.inc  
  
ret_val = P2_RTD_Read(module, channel)
```

### Parameters

module	Specified module address (1...15).	LONG
channel	Number (1...8) of the measuring channel.	LONG
ret_val	Current temperature value (24 Bit) in digits.	LONG

### Notes

With "single shot" mode a measurement value may only be read, if the measurement cycle is completed (see **P2\_RTD\_Status**).

### See also

[P2\\_RTD\\_Channel\\_Config](#), [P2\\_RTD\\_Config](#), [P2\\_RTD\\_Convert](#), [P2\\_RTD\\_Read8](#), [P2\\_RTD\\_Start](#), [P2\\_RTD\\_Status](#)

### Valid for

RTD-8 Rev. E

### Example

- / -



**P2\_RTD\_Read8** returns the current digital measurement values of temperature sensors at all channels on the module.

## Syntax

```
#Include ADwinPro_All.inc

P2_RTD_Read8(module, array[], index)
```

## Parameters

<code>module</code>	Specified module address (1...15).	LONG
<code>array[ ]</code>	Destination array, where measurement values are stored.	ARRAY LONG
<code>index</code>	Index of the array element where the first measurement value is stored.	LONG

## Notes

8 measurement values will be stored in the destination array even when you have enabled less than 8 measurement channels. The measurement values are stored with ascending channel number.

## See also

[P2\\_RTD\\_Channel\\_Config](#), [P2\\_RTD\\_Config](#), [P2\\_RTD\\_Convert](#), [P2\\_RTD\\_Read](#), [P2\\_RTD\\_Start](#), [P2\\_RTD\\_Status](#)

## Valid for

RTD-8 Rev. E

## Example

see [P2\\_RTD\\_Start](#)

## P2\_RTD\_Read8

## P2\_RTD\_Start

**P2\_RTD\_Start** starts the temperature measurement cycle on all specified modules at the same time.

### Syntax

```
#Include ADwinPro_All.inc

P2_RTD_Start(module_pattern)
```

### Parameters

**module\_**  
**pattern** Bit pattern to select the module addresses, where LONG  
a measurement cycle is started:  
Bit = 0: Ignore module address.  
Bit = 1: Select module address.

Bits in <b>pattern</b>	31:15	14	13	...	01	00
Module address	–	15	14	...	2	1

### Notes

Before starting a measurement cycle you have to set the operating modes of the modules with **P2\_RTD\_Config** and the operating modes of each channel separately with **P2\_RTD\_Config\_Channel**.

If a module without temperature measurement is selected, the instruction may cause unpredictable consequences.

### See also

[P2\\_RTD\\_Channel\\_Config](#), [P2\\_RTD\\_Config](#), [P2\\_RTD\\_Convert](#), [P2\\_RTD\\_Read](#), [P2\\_RTD\\_Read8](#), [P2\\_RTD\\_Status](#)

### Valid for

RTD-8 Rev. E

### Example

```
#Include ADwinPro_All.inc
#Define module 2

Dim values24[8] As Long
Dim channel As Long
Dim i As Long
Dim run_state As Long
Dim status As Long

Init:
P2_RTD_Config(module, 0, 0) 'single shot mode
Rem use channels 1...6
For i = 1 To 6
    Rem do channel settings: 3 wire, PT100, 50 Hz filter
    P2_RTD_Channel_Config(module, i, 1, 2, 0, 7, 50)
Next
Processdelay=50000000
run_state = 0

Event:
SelectCase run_state
Case 0
    Rem start measurement cycle
    P2_RTD_start(Shift_Left(1, module-1))
    run_state = 1
Case 1
    Rem check for end of measurement cycle
    status = P2_RTD_status(module)
    If (status = 0) Then run_state = 2
Case 2
    Rem read measured values and prepare start of next cycle
    P2_RTD_read8(module, values24, 1) 'messwerte lesen
    For i = 1 To 6
        Rem convert measurement values
        fpar[i] = P2_RTD_convert(values24[i], 0, 1)
    Next
    run_state = 0
EndSelect
```

## P2\_RTD\_Status

**P2\_RTD\_Status** returns the status of temperature measurement cycle in "single shot" mode on the specified module.

### Syntax

```
#Include ADwinPro_All.inc  
  
ret_val = P2_RTD_Status(module)
```

### Parameters

<code>module</code>	Specified module address (1...15).	LONG
<code>ret_val</code>	Status of measurement cycle: 0: Measurement cycle is completed. 1: Measurement cycle is being processed.	LONG

### Notes

The instruction **P2\_RTD\_Status** is only useful with operating mode "single shot".

### See also

[P2\\_RTD\\_Channel\\_Config](#), [P2\\_RTD\\_Config](#), [P2\\_RTD\\_Convert](#), [P2\\_RTD\\_Read](#), [P2\\_RTD\\_Read8](#), [P2\\_RTD\\_Start](#)

### Valid for

RTD-8 Rev. E

### Example

see [P2\\_RTD\\_Start](#)

**P2\_TC\_Latch** copies the current voltage values at the inputs into latches.

## Syntax

```
#Include ADwinPro_All.Inc

P2_TC_Latch(module)
```

## Parameters

**module** Specified module address (1...15). LONG

## Notes

The values in the latches will be calculated into the desired return value (°C, °F, thermo voltage) when being read with **..Read\_Latch** i.

On the temperature module, the instruction **P2\_Sync\_All** has the same effect as **P2\_TC\_Latch**.

## See also

[P2\\_TC\\_Read\\_Latch](#), [P2\\_TC\\_Read\\_Latch4](#), [P2\\_TC\\_Read\\_Latch8](#),  
[P2\\_TC\\_Set\\_Rate](#), [P2\\_Sync\\_All](#)

## Valid for

TC-8-ISO Rev. E

## Example

```
#Include ADwinPro_All.Inc
```

### Init:

```
Rem Set sampling rate to 27.5 Hz
P2_TC_Set_Rate(1,8)
```

### Event:

```
Rem copy values to latches
P2_TC_Latch(1)
Rem Read temperature from channel 5, thermo couple K in °C
FPar_1 = P2_TC_Read_Latch(1,5,1,1)
```

## P2\_TC\_Latch

## P2\_TC\_Read\_Latch

**P2\_TC\_Read\_Latch** returns the thermo voltage ( $\mu\text{V}$ ) or temperature ( $^{\circ}\text{C}$  /  $^{\circ}\text{F}$ ) of the selected channel of the module.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_TC_Read_Latch(module, channel,
                           tc_element, ret_type)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>channel</b>	Number (1...8) of the channel being read.	LONG
<b>tc_element</b>	Type of thermo couple: -1: no conversion, thus current thermo voltage without cold junction compensation. 0: Thermo couple type J 1: Thermo couple type K 2: Thermo couple type N 3: Thermo couple type S 4: Thermo couple type T 5: Thermo couple type R 6: Thermo couple type E 7: Thermo couple type B	LONG
<b>ret_type</b>	Type of return value <b>ret_val</b> : 0: Thermo voltage in $\mu\text{V}$ ; with <b>tc_element</b> = -1 without cold junction compensation. 1: Temperature in $^{\circ}\text{C}$ . 2: Temperature in $^{\circ}\text{F}$ .	LONG
<b>ret_val</b>	Measurement value of the channel, depending on <b>ret_type</b> and <b>tc_element</b> .	FLOAT

### Notes

The module regularly samples the channels (setting the sample rate see **P2\_TC\_Set\_Rate**). The instruction **P2\_TC\_Read\_Latch** returns the most recently sampled value.

If you want to read several channels using the same thermo couple type, the instructions **P2\_TC\_Read\_Latch4** and **P2\_TC\_Read\_Latch8** are a lot faster.

We recommend to use a constant for **tc\_element**. Using a variable the instruction requires much more program memory.

If you set **tc\_element** = -1, the thermo voltage value will be left unchanged, i.e. neither a cold junction correction is performed nor the thermo couple characteristics are considered.

The value range of **ret\_val** will then be -80000 $\mu\text{V}$ ...80000 $\mu\text{V}$ .

The thermoelectric voltage and the temperature values in  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$  apply only for standard thermocouples according to the norm IEC 584-1. The value ranges are:

Type	Temperature range [ $^{\circ}\text{C}$ ]	Temperature range [ $^{\circ}\text{F}$ ]	Thermo voltage [ $\mu\text{V}$ ]
B	250...1820	482...3329.6	291...13820
E	-200...1000	-328...1832	-8825...76373
J	-210...1200	-346...2192	-8095...69553

Type	Temperature range [°C]	Temperature range [°F]	Thermo voltage [μV]
K	-200...1372	-328...2501.6	-5891...54886
N	-200...1300	-328...2372	-3990...47513
R	-50...1768	-58...3214.4	-226...21101
S	-50...1768	-58...3214.4	-236...18693
T	-200...400	-454...752	-5603...20872

### See also

[P2\\_TC\\_Latch](#), [P2\\_TC\\_Read\\_Latch4](#), [P2\\_TC\\_Read\\_Latch8](#), [P2\\_TC\\_Set\\_Rate](#)

### Valid for

TC-8-ISO Rev. E

### Example

```
#Include ADwinPro_All.Inc
```

#### Init:

```
Rem Set sampling rate to 27.5 Hz
P2_TC_Set_Rate(1,8)
```

#### Event:

```
Rem copy values to latches
P2_TC_Latch(1,8)
Rem Read temperature from channel 5, thermo couple K in °C
FPar_1 = P2_TC_Read_Latch(1,5,1,1)
```

## P2\_TC\_Read\_Latch4

**P2\_TC\_Read\_Latch4** returns the thermo voltage ( $\mu\text{V}$ ) or temperature ( $^{\circ}\text{C}$  /  $^{\circ}\text{F}$ ) of the channels 1...4 of the module.

### Syntax

```
#Include ADwinPro_All.Inc

P2_TC_Read_Latch4(module, tc_element, ret_type,
                  array[], array_start_index)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>tc_element</b>	Type of thermo couple: -1: no conversion, thus current thermo voltage without cold junction compensation. 0: Thermo couple type J 1: Thermo couple type K 2: Thermo couple type N 3: Thermo couple type S 4: Thermo couple type T 5: Thermo couple type R 6: Thermo couple type E 7: Thermo couple type B	LONG
<b>ret_type</b>	Type of return value <b>ret_val</b> : 0: Thermo voltage in $\mu\text{V}$ ; with <b>tc_element</b> = -1 without cold junction compensation. 1: Temperature in $^{\circ}\text{C}$ . 2: Temperature in $^{\circ}\text{F}$ .	LONG
<b>array[]</b>	Destination array to store measurement value of channels 1...4; measurement values depending on <b>ret_type</b> und <b>tc_element</b> .	ARRAY FLOAT
<b>array_start_index</b>	Index of the first array element in <b>array[]</b> being written.	LONG

### Notes

The module regularly samples the channels (setting the sample rate see **P2\_TC\_Set\_Rate**). The instruction **P2\_TC\_Read\_Latch4** returns the most recently sampled values of the channels 1...4.

We recommend to use a constant for **tc\_element**. Using a variable the instruction requires much more program memory.

If you set **tc\_element** = -1, the thermo voltage value will be left unchanged, i.e. neither a cold junction correction is performed nor the thermo couple characteristics are considered.

The value range of **ret\_val** will then be  $-80000\mu\text{V} \dots 80000\mu\text{V}$ .

The thermoelectric voltage and the temperature values in  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$  apply only for standard thermocouples according to the norm IEC 584-1. The value ranges are:

Type	Temperature range [ $^{\circ}\text{C}$ ]	Temperature range [ $^{\circ}\text{F}$ ]	Thermo voltage [ $\mu\text{V}$ ]
B	250...1820	482...3329.6	291...13820
E	-200...1000	-328...1832	-8825...76373
J	-210...1200	-346...2192	-8095...69553



Type	Temperature range [°C]	Temperature range [°F]	Thermo voltage [μV]
K	-200...1372	-328...2501.6	-5891...54886
N	-200...1300	-328...2372	-3990...47513
R	-50...1768	-58...3214.4	-226...21101
S	-50...1768	-58...3214.4	-236...18693
T	-200...400	-454...752	-5603...20872

### See also

[P2\\_TC\\_Latch](#), [P2\\_TC\\_Read\\_Latch](#), [P2\\_TC\\_Read\\_Latch8](#), [P2\\_TC\\_Set\\_Rate](#)

### Valid for

TC-8-ISO Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim cnt As Long
Dim values[1000] As Float
```

#### Init:

```
Rem Set sampling rate to 27.5 Hz
P2_TC_Set_Rate(1,8)
cnt = 1
```

#### Event:

```
Rem copy values to latches
P2_TC_Latch(1)
Rem Read temperature from channels 1..4, thermo couple J in °F
P2_TC_Read_Latch4(1,0,2,values,cnt)
Rem increase counter
cnt = cnt + 4 : If (cnt > 1000) Then cnt = 1
```

## P2\_TC\_Read\_Latch8

**P2\_TC\_Read\_Latch8** returns the thermo voltage ( $\mu\text{V}$ ) or temperature ( $^{\circ}\text{C}$  /  $^{\circ}\text{F}$ ) of the channels 1...8 of the module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_TC_Read_Latch8(module, tc_element, ret_type,  
array[], array_start_index)
```

### Parameters

module	Specified module address (1...15).	LONG
tc_element	Type of thermo couple: -1: no conversion, thus current thermo voltage without cold junction compensation. 0: Thermo couple type J 1: Thermo couple type K 2: Thermo couple type N 3: Thermo couple type S 4: Thermo couple type T 5: Thermo couple type R 6: Thermo couple type E 7: Thermo couple type B	LONG
ret_type	Type of return value <b>ret_val</b> : 0: Thermo voltage in $\mu\text{V}$ ; with <b>tc_element</b> = -1 without cold junction compensation. 1: Temperature in $^{\circ}\text{C}$ . 2: Temperature in $^{\circ}\text{F}$ .	LONG
array[]	Destination array to store measurement value of channels 1...8; measurement values depending on <b>ret_type</b> und <b>tc_element</b> .	ARRAY FLOAT
array_start_index	Index of the first array element in <b>array[]</b> being written.	LONG

### Notes

The module regularly samples the channels (setting the sample rate see **P2\_TC\_Set\_Rate**). The instruction **P2\_TC\_Read\_Latch8** returns the most recently sampled values of the channels 1...8.

We recommend to use a constant for **tc\_element**. Using a variable the instruction requires much more program memory.

If you set **tc\_element** = -1, the thermo voltage value will be left unchanged, i.e. neither a cold junction correction is performed nor the thermo couple characteristics are considered.

The value range of **ret\_val** will then be  $-80000\mu\text{V}$ ... $80000\mu\text{V}$ .

The thermoelectric voltage and the temperature values in  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$  apply only for standard thermocouples according to the norm IEC 584-1. The value ranges are:

Type	Temperature range [ $^{\circ}\text{C}$ ]	Temperature range [ $^{\circ}\text{F}$ ]	Thermo voltage [ $\mu\text{V}$ ]
B	250...1820	482...3329.6	291...13820
E	-200...1000	-328...1832	-8825...76373
J	-210...1200	-346...2192	-8095...69553

Type	Temperature range [°C]	Temperature range [°F]	Thermo voltage [μV]
K	-200...1372	-328...2501.6	-5891...54886
N	-200...1300	-328...2372	-3990...47513
R	-50...1768	-58...3214.4	-226...21101
S	-50...1768	-58...3214.4	-236...18693
T	-200...400	-454...752	-5603...20872

### See also

[P2\\_TC\\_Latch](#), [P2\\_TC\\_Read\\_Latch](#), [P2\\_TC\\_Read\\_Latch4](#), [P2\\_TC\\_Set\\_Rate](#)

### Valid for

TC-8-ISO Rev. E

### Example

```
#Include ADwinPro_All.Inc
Dim cnt As Long
Dim values[1000] As Float
```

#### Init:

```
Rem Set sampling rate to 27.5 Hz
P2_TC_Set_Rate(1,8)
cnt = 1
```

#### Event:

```
Rem copy values to latches
P2_TC_Latch(1)
Rem Read temperature from channels 1..8, thermo couple J in °F
P2_TC_Read_Latch8(1,0,2,values,cnt)
Rem increase counter
cnt = cnt + 8 : If (cnt > 1000) Then cnt = 1
```

## P2\_TC\_Set\_Rate

**P2\_TC\_Set\_Rate** sets the sampling rate of the selected module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_TC_Set_Rate(module, rate)
```

### Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>rate</b>	Key figure of the specified sample rate (see table); default: 15.	LONG

Key figure	Sample rate [Hz]	ADC noise [nV]
1	3520	23000
2	1760	3500
3	880	2000
4	440	1400
5	220	1000
6	110	750
7	55	510
8	27.5	375
9	13.75	250
15	6.875	200

### Notes

The sample rate is valid for all channels in similar.

A higher sample rate refers to a higher noise signal at the ADC of the channel. The noise signal superposes the sampled signal (see table).

### See also

[P2\\_TC\\_Latch](#), [P2\\_TC\\_Read\\_Latch](#), [P2\\_TC\\_Read\\_Latch4](#), [P2\\_TC\\_Read\\_Latch8](#)

### Valid for

TC-8-ISO Rev. E

### Example

```
#Include ADwinPro_All.Inc  
  
Init:  
    Rem Set sampling rate to 27.5 Hz  
    P2_TC_Set_Rate(1,8)
```

### 3.9 Pro II: CAN bus Modules

This section describes instructions which apply to Pro II CAN bus modules:

- [CAN\\_Msg](#) (page 218)
- [P2\\_CAN\\_Interrupt\\_Source](#) (page 220)
- [P2\\_CAN\\_Set\\_LED](#) (page 222)
- [P2\\_En\\_Interrupt](#) (page 223)
- [P2\\_En\\_Receive](#) (page 225)
- [P2\\_En\\_Transmit](#) (page 226)
- [P2\\_Get\\_CAN\\_Reg](#) (page 227)
- [P2\\_Init\\_CAN](#) (page 228)
- [P2\\_Read\\_Msg](#) (page 229)
- [P2\\_Read\\_Msg\\_Con](#) (page 231)
- [P2\\_Set\\_CAN\\_Baudrate](#) (page 233)
- [P2\\_Set\\_CAN\\_Reg](#) (page 237)
- [P2\\_Transmit](#) (page 238)

In the [Instruction List sorted by Module Types](#) (annex A.2) you will find which of the functions corresponds to the *ADwin-Pro II* modules.

It is presumed that application examples use the module address 1.



## CAN\_Msg

**CAN\_Msg** is a one-dimensional array consisting of 9 elements, where the message objects of the CAN bus are saved during sending and receiving.

### Syntax

```
#Include ADwinProAll.Inc
```

```
CAN_Msg[n] = value
```

or

```
ret_val = CAN_Msg[n]
```

### Parameters

<b>n</b>	Number of an array element (1... 9).	LONG
<b>value</b>	Expression whose value (0...256) is written into the message object.	LONG
<b>ret_val</b>	Value (0...256), which is read from the message object.	LONG

### Notes

The first 8 elements of the array contain the data bytes 1...8 and the 9th array element the amount of valid data bytes of the message. Here a value from 0 to 8 must be entered.

The data bytes use only the bits 7...0 in the array elements, bits 31...8 are ignored.

The values in the array **CAN\_Msg[ ]** must be entered before executing **P2\_Transmit**.

### See also

[P2\\_En\\_Receive](#), [P2\\_En\\_Transmit](#), [P2\\_Read\\_Msg](#), [P2\\_Transmit](#)

### Valid for

CAN-2 Rev. E

## Example

*REM Sends a 32 Bit FLOAT-value (here: Pi) as sequence of  
REM 4 bytes in a message object  
REM (Receiving of a float value see example at [P2\\_Read\\_Msg](#))*

```
#Include ADwinPro_All.Inc
#Define pi 3.14159265
Dim i As Long

Init:
    P2_Init_CAN(1,1)          'Initialize CAN controller 1

    REM Enable message object 6 of controller 1
    REM for sending with the identifier 40 (11 bit)
    P2_En_Transmit(1,1,6,40,0)

    REM Create bit pattern of Pi with data type Long
    Par_1 = Cast_FloatToLong(pi)

    REM divide bit pattern (32 Bit) into 4 bytes
    CAN_Msg[4] = Par_1 And 0FFh 'assign LSB
    For i = 1 To 3
        CAN_Msg[4-i] = Shift_Right(Par_1,8*i) And 0FFh
    Next i
    CAN_Msg[9] = 4             'message length in bytes

Event:
    P2_Transmit(1,1,6)         'Send the message object 6
```

## P2\_CAN\_Interrupt\_Source

**P2\_CAN\_Interrupt\_Source** returns the CAN channels which have generated an event signal (interrupt).

### Syntax

```
#INCLUDE ADwinPro_All.inc  
  
ret_val = P2_CAN_Interrupt_Source(module)
```

### Parameters

<code>module</code>	Eingestellte Moduladresse (1...15).	LONG
<code>ret_val</code>	Bitmuster, das die Interrupt-Quelle angibt.	LONG

Bit-Nr.	31...2	1	0
CAN-Kanal	–	2	1

### Notes

The instruction is useful only if generating event signals is enabled with **P2\_En\_Interrupt**.

**P2\_CAN\_Interrupt\_Source** runs much faster than reading the interrupt register on a CAN controller.

After an event signal having been generated the message of the generating message object must be read with **P2\_Read\_Msg**, thus enabling the message object to generate a new event signal. In the meanwhile the CAN controller ignores incoming messages for this message object.

### See also

[P2\\_En\\_Interrupt](#), [P2\\_Init\\_CAN](#), [P2\\_Read\\_Msg](#)

### Valid for

CAN-2 Rev. E



## Example

```
#Include ADwinPRO_ALL.INC
```

### Init:

```
P2_Init_CAN(1,1)           'initialize channel 1
P2_En_Receive(1,1,3,1,0) 'configure msg objects 3 and 15
P2_En_Receive(1,1,15,385,0) 'for read
P2_En_Interrupt(1,1,3)     'configure msg objects 3 and 15
P2_En_Interrupt(1,1,15)   'for interrupt
P2_Event_Enable(1,1)      'enable event interrupt
```

### Event:

```
Par_13 = P2_CAN_Interrupt_Source(1) 'check for interrupt
If (Par_13 And 01b = 1) Then
    Par_14 = CAN_Interrupt_Msg(1,1) 'get interrupting msg object
    Rem get msg object = enable new interrupt
    Par_15 = P2_Read_Msg(1,1,CAN_Interrupt_Msg(1,1))
EndIf
```

### Function CAN\_Interrupt\_Msg(module,channel) As Long

```
REM read interrupt register and change value to objekt no.
CAN_Interrupt_Msg = P2_Get_CAN_Reg(module,channel,5fh)
If (CAN_Interrupt_Msg = 2) Then
    CAN_Interrupt_Msg = 15
Else
    CAN_Interrupt_Msg = CAN_Interrupt_Msg - 2
EndIf
EndFunction
```

The value of the interrupt register **5Fh** refers to a message object according to the following table:

Register value	2	3	4	...	16
No. of message object	15	1	2	...	14

## P2\_CAN\_Set\_LED

**P2\_CAN\_Set\_LED** switches the additional LED of a CAN channel on (with color) or off.

### Syntax

```
#include ADwinPro_All.inc  
  
P2_CAN_Set_LED(module, channel, led_col)
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
<b>led_col</b>	Status and Farbe of the additional LED: 0: LED off. 1: LED on, color red. 2: LED on, color green. 3: LED on, color orange.	LONG

### Notes

- / -

### See also

[P2\\_Set\\_LED](#)

### Valid for

CAN-2 Rev. E

### Example

```
#include ADwinPro_All.inc  
Init:  
    P2_Init_CAN(1,1)           'initialize CAN controller  
    P2_CAN_Set_LED(1,1,2)      'switch on channel 1 LED, color green
```

**P2\_En\_Interrupt** configures a message object of the specified module to generate an event signal (interrupt) when a message arrives.

## Syntax

```
#Include ADwinPro_All.inc
```

```
P2_En_Interrupt (module, channel, msg_no)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
<b>msg_no</b>	Number (1...15) of the message object in the CAN controller.	LONG

## Notes

A generated event signal will be forwarded to the processor module only, when the event signal is enabled with **P2\_Event\_Enable**. The specified message objects must be configured at first before the event signal is enabled at last.

In a system only one event input may be active, in addition to a processor module, that is you have to disable an actually active event input, before you enable the event input of another module.

After an event signal having been generated the message of the generating message object must be read with **P2\_Read\_Msg**, thus enabling the message object to generate a new event signal. In the meanwhile the CAN controller ignores incoming messages for this message object.

## See also

[P2\\_CAN\\_Interrupt\\_Source](#), [P2\\_En\\_Receive](#), [P2\\_Event\\_Enable](#), [P2\\_Event\\_Read](#), [P2\\_Get\\_CAN\\_Reg](#), [P2\\_Init\\_CAN](#)

## Valid for

CAN-2 Rev. E

## P2\_En\_Interrupt



### Example

```
#Include ADwinPro_All.Inc

Init:
    REM Initialize CAN controller 1 on the CAN module 1
    P2_Init_CAN(1,1)
    REM Configure message objects 3 and 15 for read
    P2_En_Receive(1,1,3,1,0)
    P2_En_Receive(1,1,15,385,0)
    REM Configure interrupt for message objects 3 and 15
    P2_En_Interrupt(1,1,3)
    P2_En_Interrupt(1,1,15)
    REM Enable event signal
    P2_Event_Enable(1,1)
Event:
    REM Read interrupt register (see below)
    Par_13 = P2_Get_CAN_Reg(1,1,5Fh)
    REM Convert register value into no. of object
    If (Par_13 = 2) Then
        Par_13 = 15
    Else
        Par_13 = Par_13 - 2
    EndIf
    Rem get msg object = enable new interrupt
    Par_15 = P2_Read_Msg(1,1,Par_13)
```

The value of the interrupt register **5Fh** refers to a message object according to the following table:

Register value	2	3	4	...	16
No. of message object	15	1	2	...	14

**P2\_En\_Receive** enables a message object on the specified module to receive messages.

For the message object the CAN channel, the length of the message identifier and the identifier itself are determined.

## Syntax

```
#Include ADwinPro_All.inc
```

```
P2_En_Receive(module, channel, msg_no, id, id_extend)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
<b>msg_no</b>	Number (1...15) of the message object in the CAN controller.	LONG
<b>id</b>	Identifier of the messages which are to be received in this message object ( $0 \dots 2^{11}$ or $0 \dots 2^{29}$ ).	LONG
<b>id_extend</b>	Marker for the length of the identifier: 0: 11 bit identifier. 1: 29 bit identifier.	LONG

## Notes

A message object is only able to receive messages from the CAN bus, when it has been enabled before by **P2\_En\_Receive**.

## See also

[CAN\\_Msg](#), [P2\\_En\\_Transmit](#), [P2\\_Init\\_CAN](#), [P2\\_Read\\_Msg](#), [P2\\_Transmit](#)

## Valid for

CAN-2 Rev. E

## Example

```
#Include ADwinPro_All.inc
```

### Init:

```
REM Initialize CAN controller 1 on CAN module 1
P2_Init_CAN(1,1)
REM Enable message object 1 to receive CAN messages
REM with the 11 bit-identifier 200
P2_En_Receive(1,1,1,200,0)
```

## P2\_En\_Receive

## P2\_En\_Transmit

**P2\_En\_Transmit** enables a message object on the specified module to transmit messages.

The CAN channel, the length of the message identifier and the identifier itself are determined for the message object.

### Syntax

```
#Include ADwinPro_All.inc
```

```
P2_En_Transmit(module, channel, msg_no, id, id_extend)
```

### Parameters

<code>module</code>	Selected module address (1...15).	LONG
<code>channel</code>	Number (1, 2) of the CAN channel that specifies the CAN controller.	LONG
<code>msg_no</code>	Number (1...14) of the message object in the CAN controller.	LONG
<code>id</code>	Identifier of the messages that are sent in this message object (0...2 <sup>11</sup> oder 0...2 <sup>29</sup> ).	LONG
<code>id_extend</code>	Marker for the length of the identifier: 0: 11 bit identifier. 1: 29 bit identifier.	LONG

### Notes

A message object can only transmit messages to the CAN bus when it has been enabled before by **P2\_En\_Transmit**.

### See also

[CAN\\_Msg](#), [P2\\_En\\_Receive](#), [P2\\_Init\\_CAN](#), [P2\\_Read\\_Msg](#), [P2\\_Transmit](#)

### Valid for

CAN-2 Rev. E

### Example

```
#Include ADwinPro_All.inc
```

```
Init:
```

```
REM Initialize CAN controller 1 on CAN module 1
```

```
P2_Init_CAN(1,1)
```

```
REM Enable message object 6 for sending of CAN messages
```

```
REM with the 11 bit-identifier 40
```

```
P2_En_Transmit(1,1,6,40,0)
```

**P2\_Get\_CAN\_Reg** returns the contents of a specified register on a CAN controller on the specified module.

## Syntax

```
#Include ADwinPro_All.inc

ret_val = P2_Get_CAN_Reg(module, channel, regno)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
<b>regno</b>	Register number of the CAN controller (0...255).	LONG
<b>ret_val</b>	Contents of the CAN controller register.	LONG

## Notes

The register number corresponds to the register number of the CAN controller (see data-sheet AN82527 from Intel<sup>®</sup>), e.g.:

- address **00h**: control register
- address **01h**: status register
- address **5fh**: interrupt register

## See also

[P2\\_En\\_Interrupt](#), [P2\\_Init\\_CAN](#), [P2\\_Set\\_CAN\\_Baudrate](#), [P2\\_Set\\_CAN\\_Reg](#)

## Valid for

CAN-2 Rev. E

## Example

```
#Include ADwinPro_All.inc
Init:
    REM Initialize CAN controller 1 on CAN module 1
    P2_Init_CAN(1,1)
    REM Read out the control register of CAN controller 1, module 1
    Par_1 = P2_Get_CAN_Reg(1,1,0)
```

## P2\_Get\_CAN\_Reg

## P2\_Init\_CAN

**P2\_Init\_CAN** initializes one of the CAN controllers on the specified module and sets it into an initial status.

### Syntax

```
#Include ADwinPro_All.inc  
P2_Init_CAN(module, channel)
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG

### Notes:

The instruction executes the following actions:

- Reset (hardware reset of the CAN controller).
- All filters are set to "must match".
- Set clockout register to 0 (= external frequency will not be divided).
- Set bus configuraton register to 0
- Set transfer rate for the CAN bus to 1MBit/s.
- Disable all message objects.

This instruction must be executed at the beginning of the process (if possible in the process sections **LowInit:** or **Init:**) before other instructions access the CAN controller.

With Low speed CAN the maximum transfer rate is 125kBit/s and therefore must be newly set with **P2\_Set\_CAN\_Baudrate**.

### See also

[P2\\_En\\_Receive](#), [P2\\_En\\_Transmit](#), [P2\\_Get\\_CAN\\_Reg](#), [P2\\_Set\\_CAN\\_Baudrate](#), [P2\\_Set\\_CAN\\_Reg](#)

### Valid for

CAN-2 Rev. E

### Example

```
#Include ADwinPro_All.inc  
Init:  
    REM Initialize CAN controller 1 on CAN module 1  
    P2_Init_CAN(1,1)
```





**P2\_Read\_Msg** returns the information if a new message in a message object of one of the CAN controllers on the module has been received.

If yes, the message is copied to the array **CAN\_Msg** [ ] and the identifier is returned.

## Syntax

```
#Include ADwinPro_All.inc

ret_val = P2_Read_Msg(module, channel, msg_no)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
<b>msg_no</b>	Number (1... 15) of the message object in the CAN controller.	LONG
<b>ret_val</b>	≥-1: A new message has arrived, the value is the identifier of the message object. -1: No new message has arrived.	LONG

## Notes

To receive a message you have to follow the correct order:

- Once: Enable the message object with **P2\_En\_Receive** for receiving.
- As often as needed: Check for a received message and save to **CAN\_Msg** with **P2\_Read\_Msg**.

You can read a received message only once.

## See also

[CAN\\_Msg](#), [P2\\_En\\_Receive](#), [P2\\_En\\_Transmit](#), [P2\\_Init\\_CAN](#), [P2\\_Transmit](#)

## Valid for

CAN-2 Rev. E

## P2\_Read\_Msg

### Example

*REM If a new message with the correct identifier is received  
REM the data is read out. The first 4 bytes of the message are  
REM combined to a float value of length 32 bit. (Sending a  
REM float value see example of P2\_Transmit).*

```
#Include ADwinPro_All.Inc
Dim n As Long
```

#### Init:

```
Par_1 = 0
P2_Init_CAN(1,1)           'Initialize CAN controller 1
P2_En_Receive(1,1,8,40,0) 'Initialize the message object 8
                           'to receive CAN messages with
                           'identifier 40
```

#### Event:

*REM If the message is changed, read out the received data  
REM from object 8 and save the identifier to parameter 9.  
REM The data bytes are in the array CAN\_MSG[].*

```
Par_9 = Read_Msg(1,1,8)
```

```
If (Par_9 = 40) Then
```

*REM New message for message object with the identifier 40  
REM has arrived*

```
Par_1 = CAN_Msg[1]           'Read out high-byte
```

```
For n = 2 To 4               'Combine with remaining 3 bytes to
```

```
Par_1 = Shift_Left(Par_1,8) + CAN_Msg[n] 'a 32-bit value
```

```
Next n
```

*REM Convert the bit pattern in Par\_1 to data type FLOAT and  
REM assign to the variable FPar\_1.*

```
FPar_1 = Cast_LongToFloat(Par_1)
```

```
EndIf
```

**P2\_Read\_Msg\_Con** returns the information if a new message in a message object of one of the CAN controllers on the module has been received. If yes, the message is copied to the array **CAN\_Msg[ ]** and the identifier is returned.

## Syntax

```
#Include ADwinPro_All.inc

ret_val = P2_Read_Msg_Con(module, channel, msg_no)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
<b>msg_no</b>	Number (1... 15) of the message object in the CAN controller.	LONG
<b>ret_val</b>	≥-1: A new message has arrived, the value is the identifier of the message object. -1: No new message has arrived.	LONG

## Notes

To receive a message you have to follow the correct order:

- Once: Enable the message object with **P2\_En\_Receive** for receiving.
- As often as needed: Check for a received message and save to **CAN\_Msg** with **P2\_Read\_Msg\_Con**.

You can read a received message only once.

## See also

[CAN\\_Msg](#), [P2\\_En\\_Receive](#), [P2\\_En\\_Transmit](#), [P2\\_Init\\_CAN](#), [P2\\_Transmit](#)

## Valid for

CAN-2 Rev. E

## P2\_Read\_Msg\_Con

### Example

*REM If a new message with the correct identifier is received  
REM the data is read out. The first 4 bytes of the message are  
REM combined to a float value of length 32 bit. (Sending a  
REM float value see example of [P2\\_Transmit](#)).*

```
#Include ADwinPro_All.Inc
Dim n As Long
```

#### Init:

```
Par_1 = 0
P2_Init_CAN(1,1)           'Initialize CAN controller 1
P2_En_Receive(1,1,8,40,0) 'Initialize the message object 8
                           'to receive CAN messages with
                           'identifier 40
```

#### Event:

*REM If the message is changed, read out the received data  
REM from object 8 and save the identifier to parameter 9.  
REM The data bytes are in the array CAN\_MSG[].*

```
Par_9 = Read_Msg_Con(1,1,8)
```

```
If (Par_9 = 40) Then
```

*REM New message for message object with the identifier 40  
REM has arrived*

```
Par_1 = CAN_Msg[1]           'Read out high-byte
```

```
For n = 2 To 4               'Combine with remaining 3 bytes to
```

```
Par_1 = Shift_Left(Par_1,8) + CAN_Msg[n]'a 32-bit value
```

```
Next n
```

*REM Convert the bit pattern in Par\_1 to data type FLOAT and  
REM assign to the variable FPar\_1.*

```
FPar_1 = Cast_LongToFloat(Par_1)
```

```
EndIf
```

**P2\_Set\_CAN\_Baudrate** sets the baud rate on one of the controllers on the specified module and returns the status information.

## Syntax

```
#Include ADwinPro_All.inc

ret_val = P2_Set_CAN_Baudrate(module, channel, rate)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Number (1, 2) of the CAN channel, that determines the CAN controller.	LONG
<b>rate</b>	Baud rate of the CAN controller: High speed CAN: 5000...1000000 Bit/s. Low speed CAN: 5000 ... 125000 Bit/s.	FLOAT
<b>ret_val</b>	Status of the instruction: 0: Baud rate is set. 1: Baud rate is not allowed and cannot be set.	LONG

## Notes

The available baud rates (bus frequencies) are given in the table "[Available Baud rates](#)". Please use the table's notation exactly, i.e. non-integer baud rates with 4 decimal places; values with different notation will be rejected as not allowed.

The instruction executes the following actions:

- Checks if the transferred Baud rate is allowed. If not then set the return value to 1 and stop processing.
- Set the registers of the CAN controller for the Baud rate.
- Set sampling mode to 0: One sample per bit.
- Select the settings in such a way that the sample point is always between 60% and 72% of the total bit length.
- Set the jump width for synchronization to 1.

In special cases it may be of interest to set a baud rate in a different way than the instruction works. The manual "Pro hardware" gives an explanation how to do this.

The instruction should be called in the program sections **LowInit:** or **Init:**, after the instruction **P2\_Init\_CAN**, because otherwise the set Baud rate will be overwritten by the default setting (1MBit/s).

## See also

[P2\\_Get\\_CAN\\_Reg](#), [P2\\_Init\\_CAN](#), [P2\\_Set\\_CAN\\_Reg](#)

## Valid for

CAN-2 Rev. E

## Example

```
#Include ADwinPro_All.inc
Dim status As Long
Init:
    P2_Init_CAN(1,1)          'Initialize CAN controller
    status = P2_Set_CAN_Baudrate(1,1,125000) 'set rate 125 kBit/s
```

## P2\_Set\_CAN\_Baudrate



### Available Baud rates

Available Baud rates [Bit/s]				
1000000.0000	888888.8889	800000.0000	727272.7273	666666.6667
615384.6154	571428.5714	533333.3333	500000.0000	470588.2353
444444.4444	421052.6316	400000.0000	380952.3810	363636.3636
347826.0870	333333.3333	320000.0000	307692.3077	296296.2963
285714.2857	266666.6667	250000.0000	242424.2424	235294.1176
222222.2222	210526.3158	205128.2051	200000.0000	190476.1905
181818.1818	177777.7778	173913.0435	166666.6667	160000.0000
156862.7451	153846.1538	148148.1481	145454.5455	142857.1429
140350.8772	133333.3333	126984.1270	125000.0000	123076.9231
121212.1212	117647.0588	115942.0290	114285.7143	111111.1111
106666.6667	105263.1579	103896.1039	102564.1026	100000.0000
98765.4321	95238.0952	94117.6471	90909.0909	88888.8889
87912.0879	86956.5217	84210.5263	83333.3333	81632.6531
80808.0808	80000.0000	78431.3725	76923.0769	76190.4762
74074.0741	72727.2727	71428.5714	70175.4386	69565.2174
68376.0684	67226.8908	66666.6667	66115.7025	64000.0000
63492.0635	62500.0000	61538.4615	60606.0606	60150.3759
59259.2593	58823.5294	57971.0145	57142.8571	55944.0559
55555.5556	54421.7687	53333.3333	52631.5789	52287.5817
51948.0519	51282.0513	50000.0000	49689.4410	49382.7160
48484.8485	47619.0476	47337.2781	47058.8235	46783.6257
45714.2857	45454.5455	44444.4444	43956.0440	43478.2609
42780.7487	42328.0423	42105.2632	41666.6667	41025.6410
40816.3265	40404.0404	40000.0000	39215.6863	38647.3430
38461.5385	38277.5120	38095.2381	37037.0370	36363.6364
36199.0950	35714.2857	35555.5556	35087.7193	34782.6087
34632.0346	34482.7586	34188.0342	33613.4454	33333.3333
33057.8512	32921.8107	32388.6640	32258.0645	32000.0000
31746.0317	31620.5534	31372.5490	31250.0000	30769.2308
30651.3410	30303.0303	30075.1880	29629.6296	29411.7647
29304.0293	29090.9091	28985.5072	28673.8351	28571.4286
28070.1754	27972.0280	27777.7778	27681.6609	27586.2069
27210.8844	27027.0270	26936.0269	26755.8528	26666.6667
26315.7895	26143.7908	25974.0260	25806.4516	25641.0256
25396.8254	25078.3699	25000.0000	24844.7205	24767.8019
24691.3580	24615.3846	24390.2439	24242.4242	24024.0240
23809.5238	23668.6391	23529.4118	23460.4106	23391.8129
23255.8140	23188.4058	22988.5057	22857.1429	22792.0228
22727.2727	22408.9636	22222.2222	22160.6648	22038.5675
21978.0220	21739.1304	21680.2168	21621.6216	21505.3763
21390.3743	21333.3333	21276.5957	21220.1592	21164.0212
21052.6316	20833.3333	20779.2208	20671.8346	20512.8205
20460.3581	20408.1633	20202.0202	20050.1253	20000.0000
19851.1166	19753.0864	19704.4335	19656.0197	19607.8431
19512.1951	19323.6715	19230.7692	19138.7560	19047.6190

Available Baud rates [Bit/s]				
18912.5296	18867.9245	18823.5294	18648.0186	18604.6512
18518.5185	18433.1797	18390.8046	18306.6362	18181.8182
18140.5896	18099.5475	18018.0180	17857.1429	17777.7778
17738.3592	17582.4176	17543.8596	17429.1939	17391.3043
17316.0173	17241.3793	17204.3011	17094.0171	17021.2766
16949.1525	16913.3192	16842.1053	16806.7227	16771.4885
16666.6667	16632.0166	16563.1470	16528.9256	16460.9053
16393.4426	16326.5306	16260.1626	16227.1805	16194.3320
16161.6162	16129.0323	16000.0000	15873.0159	15810.2767
15779.0927	15686.2745	15625.0000	15594.5419	15503.8760
15473.8878	15444.0154	15384.6154	15325.6705	15238.0952
15180.2657	15151.5152	15122.8733	15094.3396	15065.9134
15037.5940	15009.3809	14842.3006	14814.8148	14705.8824
14652.0147	14571.9490	14545.4545	14519.0563	14492.7536
14414.4144	14336.9176	14311.2701	14285.7143	14260.2496
14184.3972	14109.3474	14035.0877	13986.0140	13937.2822
13913.0435	13888.8889	13840.8304	13793.1034	13722.1269
13675.2137	13605.4422	13582.3430	13559.3220	13513.5135
13468.0135	13445.3782	13377.9264	13333.3333	13289.0365
13223.1405	13157.8947	13136.2890	13114.7541	13093.2897
13071.8954	13008.1301	12987.0130	12903.2258	12882.4477
12820.5128	12800.0000	12759.1707	12718.6010	12698.4127
12578.6164	12558.8697	12539.1850	12500.0000	12422.3602
12403.1008	12383.9009	12345.6790	12326.6564	12307.6923
12288.7865	12195.1220	12158.0547	12121.2121	12066.3650
12030.0752	12012.0120	11994.0030	11922.5037	11904.7619
11851.8519	11834.3195	11764.7059	11730.2053	11695.9064
11661.8076	11627.9070	11611.0305	11594.2029	11544.0115
11494.2529	11477.7618	11428.5714	11396.0114	11379.8009
11363.6364	11347.5177	11299.4350	11220.1964	11204.4818
11188.8112	11111.1111	11080.3324	11034.4828	11019.2837
10989.0110	10943.9124	10928.9617	10884.3537	10869.5652
10840.1084	10810.8108	10796.2213	10781.6712	10752.6882
10695.1872	10666.6667	10638.2979	10610.0796	10582.0106
10540.1845	10526.3158	10457.5163	10430.2477	10416.6667
10389.6104	10335.9173	10322.5806	10296.0103	10269.5764
10256.4103	10230.1790	10204.0816	10101.0101	10088.2724
10062.8931	10025.0627	10012.5156	10000.0000	9937.8882
9925.5583	9876.5432	9852.2167	9828.0098	9803.9216
9791.9217	9768.0098	9756.0976	9696.9697	9685.2300
9661.8357	9615.3846	9603.8415	9569.3780	9523.8095
9456.2648	9433.9623	9411.7647	9400.7051	9367.6815
9356.7251	9324.0093	9302.3256	9291.5215	9259.2593
9227.2203	9216.5899	9195.4023	9153.3181	9142.8571
9090.9091	9070.2948	9049.7738	9039.5480	9009.0090
8958.5666	8928.5714	8918.6176	8888.8889	8879.0233

Available Baud rates [Bit/s]				
8869.1796	8859.3577	8771.9298	8743.1694	8714.5969
8695.6522	8658.0087	8648.6486	8620.6897	8602.1505
8592.9108	8556.1497	8547.0085	8510.6383	8483.5631
8474.5763	8465.6085	8456.6596	8421.0526	8403.3613
8385.7442	8333.3333	8281.5735	8264.4628	8255.9340
8230.4527	8205.1282	8196.7213	8163.2653	8130.0813
8113.5903	8105.3698	8097.1660	8088.9788	8080.8081
8064.5161	8000.0000	7976.0718	7944.3893	7936.5079
7905.1383	7843.1373	7812.5000	7804.8780	7797.2710
7774.5384	7751.9380	7736.9439	7729.4686	7714.5612
7692.3077	7662.8352	7655.5024	7619.0476	7590.1328
7575.7576	7561.4367	7547.1698	7532.9567	7518.7970
7469.6545	7441.8605	7421.1503	7407.4074	7400.5550
7386.8883	7352.9412	7326.0073	7285.9745	7272.7273
7259.5281	7246.3768	7187.7808	7168.4588	7142.8571
7136.4853	7130.1248	7111.1111	7098.4916	7092.1986
7054.6737	7017.5439	6993.0070	6956.5217	6944.4444
6926.4069	6902.5022	6896.5517	6861.0635	6820.1194
6808.5106	6802.7211	6791.1715	6779.6610	6734.0067
6688.9632	6683.3751	6666.6667	6611.5702	6578.9474
6568.1445	6562.7564	6557.3770	6535.9477	6530.6122
6493.5065	6456.8200	6451.6129	6441.2238	6410.2564
6400.0000	6379.5853	6349.2063	6324.1107	6289.3082
6274.5098	6269.5925	6250.0000	6245.1210	6211.1801
6172.8395	6163.3282	6153.8462	6144.3932	6102.2121
6060.6061	6046.8632	6037.7358	5997.0015	5961.2519
5952.3810	5925.9259	5895.3574	5865.1026	5847.9532
5818.1818	5797.1014	5772.0058	5747.1264	5714.2857
5702.0670	5681.8182	5649.7175	5614.0351	5610.0982
5555.5556	5521.0490	5517.2414	5464.4809	5434.7826
5423.7288	5376.3441	5333.3333	5291.0053	5245.9016
5208.3333	5161.2903	5079.3651	5000.0000	



**P2\_Set\_CAN\_Reg** writes a value in a register of the selected CAN controller on the specified module.

## Syntax

```
#Include ADwinPro_All.inc

P2_Set_CAN_Reg(module, channel, regno, value)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
<b>regno</b>	Register number (0...255) of the CAN controller.	LONG
<b>value</b>	Value (0...255), written into the controller register.	LONG

## Notes

The register number which has to be indicated corresponds to the register number of the CAN controller (see data-sheet AN82527 from Intel®). For instance the controll register has the address 0 and the status register the address 1.

## See also

[P2\\_Init\\_CAN](#), [P2\\_Set\\_CAN\\_Baudrate](#), [P2\\_Get\\_CAN\\_Reg](#)

## Valid for

CAN-2 Rev. E

## Example

```
#Include ADwinPro_All.inc
Init:
P2_Init_CAN(1,1)           'Initialize CAN controller
P2_Set_CAN_Reg(1,1,0,1)    'Set control register to the value 1
```

## P2\_Set\_CAN\_Reg

## P2\_Transmit

**P2\_Transmit** reads the data from the array **CAN\_Msg**. As soon as the message object in one of the CAN controllers has access rights to the CAN bus, the message is sent.

### Syntax

```
#Include ADwinPro_All.inc

P2_Transmit(module, channel, msg_no)
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Number (1, 2) of the CAN channel, that determines the CAN controller.	LONG
<b>msg_no</b>	Number (1... 14) of the message object in the CAN controller.	LONG

### Notes

This instruction can only be executed when the corresponding message object has been configured before to send with the **P2\_En\_Transmit**.

The message data must be entered in **CAN\_Msg[]**, before executing **P2\_Transmit**.

### See also

[CAN\\_Msg](#), [P2\\_En\\_Receive](#), [P2\\_En\\_Transmit](#), [P2\\_Read\\_Msg](#)

### Valid for

CAN-2 Rev. E

### Example

*REM Sends a 32 Bit FLOAT-value (here: Pi) as sequence of  
REM 4 bytes in a message object  
REM (Receiving of a float value see example at [P2\\_Read\\_Msg](#))*

```
#Include ADwinPro_All.inc
#Define pi 3.14159265
Dim i As Long

Init:
    P2_Init_CAN(1,1)           'Initialize CAN controller 1

    REM Enable message object 6 of controller 1
    REM for sending with the identifier 40 (11 bit)
    P2_En_Transmit(1,1,6,40,0)

    REM Create bit pattern of Pi with data type Long
    Par_1 = Cast_FloatToLong(pi)

    REM divide bit pattern (32 Bit) into 4 bytes
    CAN_Msg[4] = Par_1 And 0FFh 'assign LSB
    For i = 1 To 3
        CAN_Msg[4-i] = Shift_Right(Par_1,8*i) And 0FFh
    Next i
    CAN_Msg[9] = 4              'message length in bytes

Event:
    P2_Transmit(1,1,6)          'Send the message object 6
```

### 3.10 Pro II: RSxxx Modules

This section describes instructions which apply to Pro II RSxxx modules:

- [P2\\_Check\\_Shift\\_Reg](#) (page 240)
- [P2\\_Get\\_RS](#) (page 241)
- [P2\\_Read\\_FIFO](#) (page 242)
- [P2\\_RS\\_Init](#) (page 243)
- [P2\\_RS\\_Reset](#) (page 245)
- [P2\\_RS485\\_Send](#) (page 246)
- [P2\\_RS\\_Set\\_LED](#) (page 247)
- [P2\\_Set\\_RS](#) (page 248)
- [P2\\_Write\\_FIFO](#) (page 249)

In the [Instruction List sorted by Module Types](#) (annex A.2) you will find which of the functions corresponds to the *ADwin-Pro II* modules.

It is presumed that application examples use the module address 1.



## P2\_Check\_Shift\_Reg

**P2\_Check\_Shift\_Reg** returns, if all data has been sent, which was written into the send-FIFO of the channel on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
ret_val = P2_Check_Shift_Reg(module, channel)
```

### Parameters

module	Selected module address (1...15).	LONG
channel	number of the channel that is to be read out (1, 2 or 1...4).	LONG
ret_val	Sending status: 0: Data has been sent (= no more data in the send-FIFO). 1: Not yet all data sent (= the send-FIFO still contains data).	LONG

### Notes

With the return value 0 both the send FIFO and the output shift register are empty. With the return value 1 there is at least one bit to be sent.

We recommend to use this instruction only after you have more experience about how the controller operates (data-sheet of the manufacturer Texas Instruments). For more common applications more comfortable instructions are available in the include file.

### See also

[P2\\_Get\\_RS](#), [P2\\_Read\\_FIFO](#), [P2\\_RS\\_Init](#), [P2\\_RS\\_Reset](#), [P2\\_RS485\\_Send](#), [P2\\_Set\\_RS](#)

### Valid for

RSxxx-2 Rev. E, RSxxx-4 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
  
Event:  
...  
Par_1 = P2_Check_Shift_Reg(1,1)'Check if channel 1 still  
                                     'has data to send  
...
```

**P2\_Get\_RS** reads out the controller register on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc
ret_val = P2_Get_RS(module,register)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>register</b>	Address of the controller register to read.	LONG
<b>ret_val</b>	Contents of the controller register.	LONG

## Notes

We recommend to use this instruction only after you have more experience about how the controller operates (data-sheet of the manufacturer Texas Instruments). For more common applications more comfortable instructions are available in the include file.

## See also

[P2\\_Check\\_Shift\\_Reg](#), [P2\\_Read\\_FIFO](#), [P2\\_RS\\_Init](#), [P2\\_RS\\_Reset](#), [P2\\_RS485\\_Send](#), [P2\\_Set\\_RS](#)

## Valid for

RSxxx-2 Rev. E, RSxxx-4 Rev. E

## Example

- / -

## P2\_Get\_RS

## P2\_Read\_FIFO

**P2\_Read\_FIFO** reads a value from the input FIFO of a specified channel on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Read_FIFO(module, channel)
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	number of the channel that is to be read out (1, 2 or 1...4).	LONG
<b>ret_val</b>	Contents of the input FIFO: -1: FIFO is empty. ≥0: Transferred value.	LONG

### Notes

- / -

### See also

[P2\\_Check\\_Shift\\_Reg](#), [P2\\_Get\\_RS](#), [P2\\_RS\\_Init](#), [P2\\_RS\\_Reset](#), [P2\\_RS485\\_Send](#), [P2\\_Set\\_RS](#)

### Valid for

RSxxx-2 Rev. E, RSxxx-4 Rev. E

### Example

```
#Include ADwinPro_All.Inc

Init:
    P2_RS_Reset(1)
    P2_RS_Init(1,1,9600,0,8,0,1)'Initialize channel 1 on module
                                '1 with 9600 Baud, without parity,
                                '8 data bits, 1 stop bit and
                                'hardware handshake (RS232 only).

Event:
    Par_1 = P2_Read_FIFO(1,1)'Get a value from the FIFO. If
                                'the FIFO is empty, -1 is returned.
```

**P2\_RS\_Init** initializes one channel on the specified module.

The following parameters are set:

- Transfer rate in Baud
- Use of test bits
- Data length
- Amount of stop bits

Transfer protocol (handshake)

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_RS_Init(module, channel, baud_rate, parity, bits,  
stop, handshake)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Channel, which is to be initialized (1, 2 or 1...4).	LONG
<b>baud_rate</b>	Transfer rate in Baud: RS232: 35 ... 115200 Baud. RS485: 35...2304000 Baud.	LONG
<b>parity</b>	Use of test bits: 0: without parity bit. 1: even parity. 2: odd parity.	LONG
<b>bits</b>	Amount of data bits (5, 6, 7 or 8).	LONG
<b>stop_bits</b>	Amount of stop bits. 0: 1 stop bit. 1: 1½ stop bits at 5 data bits; 2 stop bits at 6, 7 or 8 data bits.	LONG
<b>handshake</b>	Transfer protocol: 0: No handshake. 1: Hardware handshake (RTS/CTS), RS 232 only. 2: Software handshake (Xon/Xoff). 3: RS485.	LONG

## Notes

This instruction is necessary before working first with the selected channel, in order to set the interface parameters. They must be identical to the remote station, in order to verify a correct transfer.

The initialization is necessary after you have executed a hardware reset with **P2\_RS\_Reset**.

The baud rates are derived from the basic clock rate of 2304MHz by dividing the basic clock rate by an integer. The divisor range is 1...0FFFFh resulting into a band width of 35...2304 000 Bit/s. According to its specification, the RS-232 interface is limited to 115200 Bit/s. The following list shows some common baud rates.

Common baud rates [Bit/s]		
2304000	57600	2400
1152000	38400	1200
460800	19200	600
230400	9600	300
115200	4800	

## P2\_RS\_Init



**See also**

[P2\\_Check\\_Shift\\_Reg](#), [P2\\_Get\\_RS](#), [P2\\_Read\\_FIFO](#), [P2\\_RS\\_Reset](#),  
[P2\\_RS485\\_Send](#), [P2\\_Set\\_RS](#)

**Valid for**

RSxxx-2 Rev. E, RSxxx-4 Rev. E

**Example**

```
#include ADwinPro_All.Inc
```

**Init:**

```
P2_RS_Reset(1)           'Reset RS-module  
P2_RS_Init(1,1,9600,0,8,0,1)'Initialize channel 1 on  
                           'module 1 with 9600 Baud, without,  
                           'parity, 8 data bits, 1 stop bit and  
                           'hardware handshake (RS232 only).
```



**P2\_RS\_Reset** executes a hardware reset on the specified module and deletes the settings for all channels.

## Syntax

```
#Include ADwinPro_All.Inc

P2_RS_Reset (module)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
---------------	-----------------------------------	------

## Notes

The instruction sends a reset impulse to the input of the controller TL16C754. In the data-sheet of the controller 16C754 from Texas Instruments it is described, to which values the registers have been set after the hardware reset.

After a hardware reset an initialization with **P2\_RS\_Init** must follow, in order to initialize the controller and to set the interface parameters.

**P2\_RS\_Init** sets the same registers as a hardware reset does. Nevertheless, **P2\_RS\_Reset** should be used for the case the controller has crashed.

## See also

[P2\\_Check\\_Shift\\_Reg](#), [P2\\_Get\\_RS](#), [P2\\_Read\\_FIFO](#), [P2\\_RS\\_Init](#), [P2\\_RS485\\_Send](#), [P2\\_Set\\_RS](#)

## Valid for

RSxxx-2 Rev. E, RSxxx-4 Rev. E

## Example

```
#Include ADwinPro_All.Inc

Init:
P2_RS_Reset(1)           'Reset RS-module
P2_RS_Init(1,1,9600,0,8,0,1)'Initialize channel 1 of
                           'module 1 with 9600 Baud, without
                           'parity, 8 data bits, 1 stop bit and
                           'hardware handshake (RS232 only).
```

## P2\_RS\_Reset

## P2\_RS485\_Send

**P2\_RS485\_Send** determines the transfer direction for a specified channel on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_RS485_Send(module, channel, dir)
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Channel to be set (1, 2 or 1...4).	LONG
<b>dir</b>	Transfer direction of the channel: 0: Set channel to receive. 1: Set channel to send. 2: Set channel to send and to receive its sent data.	LONG

### Notes

Setting the transfer direction means:

- Receiver: The channel can only read data, even if data are in the output FIFO of the controller for this channel.
- Sender: The channel transfers data to the bus which are read by other devices.
- Sender/receiver: The channel can transfer data to the bus and back at the same time. Thus, the sent data can be checked.

### See also

[P2\\_Check\\_Shift\\_Reg](#), [P2\\_Get\\_RS](#), [P2\\_Read\\_FIFO](#), [P2\\_RS\\_Init](#), [P2\\_RS\\_Reset](#), [P2\\_Set\\_RS](#)

### Valid for

RSxxx-2 Rev. E, RSxxx-4 Rev. E

### Example

See Example "RS485: Receive And send" on .

**P2\_RS\_Set\_LED** switches the additional LED of a RSxxx channel on (with color) or off.

## Syntax

```
#INCLUDE ADwinPro_All.inc

P2_RS_Set_LED(module, channel, led_col)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Channel to be set (1, 2 or 1...4).	LONG
<b>led_col</b>	Status and Farbe of the additional LED: 0: LED off. 1: LED on, color red. 2: LED on, color green. 3: LED on, color orange.	LONG

## Notes

- / -

## See also

[P2\\_Set\\_LED](#)

## Valid for

RSxxx-2 Rev. E, RSxxx-4 Rev. E

## Example

```
#INCLUDE ADwinPro_All.inc

Init:
    P2_RS_Set_LED(1,1,3)      'switch on channel 1 LED, color orange
```

## P2\_RS\_Set\_LED

## P2\_Set\_RS

**P2\_Set\_RS** writes a value into a specified register on the specified module.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_Set_RS(module, register, value)
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>register</b>	Number of the register, into which data are written.	LONG
<b>value</b>	Value to be written into the register.	LONG

### Notes

We recommend to use this instruction only after you have more experience about how the controller operates (data-sheet of the manufacturer: TL16C754 from Texas Instruments). For more common applications more comfortable instructions are available in the include file.

### See also

[P2\\_Check\\_Shift\\_Reg](#), [P2\\_Get\\_RS](#), [P2\\_Read\\_FIFO](#), [P2\\_RS\\_Init](#), [P2\\_RS\\_Reset](#), [P2\\_RS485\\_Send](#)

### Valid for

RSxxx-2 Rev. E, RSxxx-4 Rev. E

### Example

- / -

**P2\_Write\_FIFO** writes a value into the send-FIFO of a specified channel on the specified module.

## Syntax

```
#Include ADwinPro_All.Inc

ret_val = P2_Write_FIFO(module, channel, value)
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>channel</b>	Channel number to whose send-FIFO data are transferred (1, 2 or 1...4).	LONG
<b>value</b>	Value to be written into the send-FIFO.	LONG
<b>ret_val</b>	Status message: 0: Data are transferred successfully. 1: Data were not transferred, send-FIFO is full.	LONG

## Notes

The instruction checks first if there is at least one memory space in the send-FIFO. If this is so, the transferred value is written into the FIFO (return value 0); otherwise a 1 is returned, indicating that the FIFO is full and writing is not possible.

The **value** to be transferred may be a single ASCII character or an ASCII instruction (chars are internally similar to Long data type). The hardware documentation contains an example for sending a string.

## See also

[P2\\_Check\\_Shift\\_Reg](#), [P2\\_Get\\_RS](#), [P2\\_Read\\_FIFO](#), [P2\\_RS\\_Init](#), [P2\\_RS\\_Reset](#), [P2\\_RS485\\_Send](#), [P2\\_Set\\_RS](#)

## Valid for

RSxxx-2 Rev. E, RSxxx-4 Rev. E

## Example

```
#Include ADwinPro_All.Inc
Dim val As Long

Init:
P2_RS_Reset(1)
P2_RS_Init(1,1,9600,0,8,0,1)'Initialize channel 1 of
                             'module 1 with 9600 Baud, no parity,
                             '8 data bits, 1 stop bit and
                             'hardware handshake (RS232 only).

Event:
Par_1 = Write_FIFO(1,1,val)'If the FIFO is not full, [val]
                             'is written into the FIFO. Otherwise
                             'a 1 in Par_1 indicates that writing
                             'into the FIFO ist not possible
                             '(FIFO full).
```

## P2\_Write\_FIFO

### 3.11 Pro II: LIN bus Interface

This section describes instructions which apply to Pro II LIN bus modules:

- [P2\\_LIN\\_Init](#) (page 251)
- [P2\\_LIN\\_Init\\_Write](#) (page 253)
- [P2\\_LIN\\_Init\\_Apply](#) (page 254)
- [P2\\_LIN\\_Reset](#) (page 255)
- [P2\\_LIN\\_Get\\_Version](#) (page 256)
- [P2\\_LIN\\_Read\\_Dat](#) (page 257)
- [P2\\_LIN\\_Msg\\_Write](#) (page 259)
- [P2\\_LIN\\_Msg\\_Transmit](#) (page 261)
- [P2\\_LIN\\_Set\\_LED](#) (page 262)

In the [Instruction List sorted by Module Types](#) (annex A.2) you will find which of the functions corresponds to the *ADwin-Pro II* modules.

It is presumed that application examples use the module address 1.



**P2\_LIN\_Init** initializes the data transfer between *ADwin* CPU and the LIN interface on a specified module.

## Syntax

```
#Include ADwinPro_All.Inc
REM define LIN settings array
Dim lin_datatable[150] As Long
ret_val = P2_LIN_Init(module, lin_datatable[])
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>lin_data- table[]</b>	Array which contains settings for data transfer between <i>ADwin</i> CPU and the LIN module.	ARRAY LONG
<b>ret_val</b>	Status of initialization: 0: initialization was successful. 1: Error: no Pro II module at this address. 2: Error: no LIN interface on the module.	LONG

## Notes

**P2\_LIN\_Init** is to be executed before data transfer between *ADwin* CPU and LIN interface. The instruction should be used in the **Init:** section.

Before initialization, an array `lin_datatable[]` with 150 elements must be declared for each module.

## Valid for

LIN-2 Rev. E

## See also

[P2\\_LIN\\_Init\\_Write](#), [P2\\_LIN\\_Init\\_Apply](#), [P2\\_LIN\\_Reset](#), [P2\\_LIN\\_Get\\_Version](#), [P2\\_LIN\\_Read\\_Dat](#), [P2\\_LIN\\_Msg\\_Write](#), [P2\\_LIN\\_Msg\\_Transmit](#)

## P2\_LIN\_Init

## Example

```
#Include ADwinPro_All.Inc

#Define mod_adr 4
Dim lin_datatable[150] As Long
Dim Data_1[20] As Long
Dim Data_2[20] As Long
Dim state As Long

Init:
    Processdelay = 30000000 '10 Hz
    'Initialize communication ADwin CPU - LIN module
    Par_1 = P2_LIN_Init(mod_adr, lin_datatable)
    If (Par_1 <> 0) Then Exit 'error
    Rem Interface 1, 9600 baud, LIN master
    P2_LIN_Init_Write(lin_datatable, 1, 9600, 1, 0)
    Rem Interface 2, 9600 baud, LIN slave
    P2_LIN_Init_Write(lin_datatable, 2, 9600, 0, 0)
    Rem message box 1 for receive on interface 2, msg id 1
    P2_LIN_Msg_Write(lin_datatable, 2, 1, 1, Data_2, 8, 0)
    P2_LIN_Init_Apply(lin_datatable)
    state = 1

Event:
    SelectCase state
        Case 1 'msg transmit
            Data_1[1] = 1
            Data_1[2] = 2
            Data_1[3] = 3
            Data_1[4] = 4
            Data_1[5] = 5
            Data_1[6] = 6
            Data_1[7] = 7
            Data_1[8] = 8
            Rem message box 1 for write on interface 1, msg id 1
            P2_LIN_Msg_Write(lin_datatable, 1, 1, 1, Data_1, 8, 1)
            Rem send header and message (interface 1 = LIN master)
            P2_LIN_Msg_Transmit(lin_datatable, 1, 1) 'msg tx
            state = 2
        Case 2 'check for msg receive, msg id 1
            P2_LIN_Read_Dat(lin_datatable, 2, 1, Data_2)
            If (Data_2[20] = 1) Then 'new msg rx
                Par_11 = Data_2[3] 'ID
                Par_12 = Data_2[4] 'Byte 1
                Par_13 = Data_2[5]
                Par_14 = Data_2[6]
                Par_15 = Data_2[7]
                Par_16 = Data_2[8]
                Par_17 = Data_2[9]
                Par_18 = Data_2[10]
                Par_19 = Data_2[11] 'Byte 8
                Par_20 = Data_2[12] 'checksum
                Par_21 = Data_2[13] 'length
                Inc Par_10
                state = 1 'new Msg tx
            EndIf
        EndSelect
```



**P2\_LIN\_Init\_Write** sets baudrate and operating mode for a specified LIN interface.

## Syntax

```
#Include ADwinPro_All.Inc

P2_LIN_Init_Write(lin_datatable[], channel,

    baudrate, mode, chs_enh)
```

## Parameters

<b>lin_data-</b> <b>table[ ]</b>	Array with settings for data transfer between ADwin CPU and the LIN module.	ARRAY LONG
<b>channel</b>	Number (1...2) of LIN interface.	LONG
<b>baudrate</b>	Baud rate (2400...19200) to run the interface with.	LONG
<b>mode</b>	Operating mode of the LIN interface: 0: Operation as LIN Slave node. 1: Operation as LIN Master node.	LONG
<b>chs_enh</b>	Type of checksum: 0: classic. 1. enhanced.	LONG

## Notes

Writing initialization data is required for each LIN interface separately. Afterwards, the initialization data must be activated for the LIN bus using **P2\_LIN\_Init\_Apply**.

If **P2\_LIN\_Init\_Write** is not processed, all interfaces run with standard settings:

- Baud rate 9600 Baud.
- Slave mode.
- Checksum type „classic“.

## Valid for

LIN-2 Rev. E

## See also

[P2\\_LIN\\_Init](#), [P2\\_LIN\\_Init\\_Apply](#), [P2\\_LIN\\_Reset](#), [P2\\_LIN\\_Get\\_Version](#),  
[P2\\_LIN\\_Read\\_Dat](#), [P2\\_LIN\\_Msg\\_Write](#), [P2\\_LIN\\_Msg\\_Transmit](#)

## Example

see [P2\\_LIN\\_Init](#)

## P2\_LIN\_Init\_Write

## P2\_LIN\_Init\_Apply

**P2\_LIN\_Init\_Apply** activates the initialization data given with **P2\_LIN\_Init\_Write** for all LIN interfaces.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_LIN_Init_Apply(lin_datatable[])
```

### Parameters

**lin\_data-** Array with settings for data transfer between **ADwin** CPU and the LIN module. **ARRAY**  
**table[]** **LONG**

### Notes

**P2\_LIN\_Init\_Apply** does not change initialization data of the LIN bus interfaces.

### Valid for

LIN-2 Rev. E

### See also

[P2\\_LIN\\_Init](#), [P2\\_LIN\\_Init\\_Write](#), [P2\\_LIN\\_Reset](#), [P2\\_LIN\\_Get\\_Version](#),  
[P2\\_LIN\\_Read\\_Dat](#), [P2\\_LIN\\_Msg\\_Write](#), [P2\\_LIN\\_Msg\\_Transmit](#)

### Example

see [P2\\_LIN\\_Init](#)

**P2\_LIN\_Reset** resets all LIN interfaces, either all settings (start-up status) or LIN-internal counters only.

## Syntax

```
#Include ADwinPro_All.Inc

P2_LIN_Reset(lin_datatable[], reset_mode)
```

## Parameters

<b>lin_data-</b> <b>table[ ]</b>	Array with settings for data transfer between ADwin CPU and the LIN module.	<b>ARRAY</b> <div>LONG</div>
<b>reset_</b> <b>mode</b>	State to set the interfaces to: 1: all settings to start-up status. 2: reset LIN counters to 0.	<div>LONG</div>

## Notes

If resetting to start-up status, the following settings are set for all LIN interfaces:

- Baud rate 9600 Baud
- Slave mode
- Set internal counters to 0 (message, timeout).

## Valid for

LIN-2 Rev. E

## See also

[P2\\_LIN\\_Init](#), [P2\\_LIN\\_Init\\_Write](#), [P2\\_LIN\\_Init\\_Apply](#), [P2\\_LIN\\_Get\\_Version](#), [P2\\_LIN\\_Read\\_Dat](#), [P2\\_LIN\\_Msg\\_Write](#), [P2\\_LIN\\_Msg\\_Transmit](#)

## Example

- / -

## P2\_LIN\_Reset

## P2\_LIN\_Get\_Version

**P2\_LIN\_Get\_Version** returns the version number of the LIN interface.

### Syntax

```
#Include ADwinPro_All.Inc  
  
ret_val = P2_LIN_Get_Version(lin_datatable[])
```

### Parameters

<code>lin_data-</code>	Array with settings for data transfer between	ARRAY
<code>table[]</code>	ADwin CPU and the LIN module.	LONG
<code>ret_val</code>	Version number (0...9999) of LIN interface.	LONG

### Notes

The version number is needed only, if you query our support about programming the LIN bus.

### Valid for

LIN-2 Rev. E

### See also

[P2\\_LIN\\_Init](#), [P2\\_LIN\\_Init\\_Write](#), [P2\\_LIN\\_Init\\_Apply](#), [P2\\_LIN\\_Reset](#),  
[P2\\_LIN\\_Read\\_Dat](#), [P2\\_LIN\\_Msg\\_Write](#), [P2\\_LIN\\_Msg\\_Transmit](#)

### Example

- / -

**P2\_LIN\_Read\_Dat** reads the data of a message box or the status of a LIN interface and writes the result into an array.

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_LIN_Read_Dat(lin_datatable[], channel, msgbox,
rd_dat[])
```

## Parameters

<b>lin_data-</b> <b>table[]</b>	Array with settings for data transfer between ADwin CPU and the LIN module.	ARRAY LONG
<b>channel</b>	Number (1...2) of LIN interface.	LONG
<b>msgbox</b>	Code number to select either a message box or LIN interface status: 1...64: Number of LIN message box, where data is read. 65: Read LIN interface status.	LONG
<b>rd_dat[]</b>	Destination array where data is returned.	ARRAY LONG

## Notes

If you provide an invalid code number **msgbox** the module may get into an instable mode. If so, you have to switch off the Pro system and restart it.

A message box / the interface status has 20 elements. The data is stored in the array elements **rd\_dat[1]** ... **rd\_dat[20]**. The following table shows the element's meaning:

Element	Message box ( <b>membox</b> =1...64)	interface status ( <b>membox</b> =65)
1	Kanalnummer (1...2)	Kanalnummer (1...2)
2	Number (1...64) of message box.	Identifier (65) for interface status.
3	Identifier (0...63) of the message box	Previously used identifier (0...63).
4	Data bytes 1...8	Number of transferred messages since xxx.
5...11		—
12	Checksum for data bytes	—
13	Number of valid data bytes	—
14	Transfer direction of message box: 0: receive 1: send	—
15	Time of LIN header in µs.	—
16	Time of LIN response in µs.	—
17	Total time of a LIN message in µs (= 15+16).	—
18	Pause time in µs between 2 data bytes. Standard: 0.	—

## P2\_LIN\_Read\_Dat

Element	Message box (membox=1...64)	interface status (membox=65)
19	Number of timeout errors for this message.	—
20	1: new message has been received. -1: no new message -2: Message is being received. -3: Message has timeout error	—

**Valid for**

LIN-2 Rev. E

**See also**

[P2\\_LIN\\_Init](#), [P2\\_LIN\\_Init\\_Write](#), [P2\\_LIN\\_Init\\_Apply](#), [P2\\_LIN\\_Reset](#),  
[P2\\_LIN\\_Get\\_Version](#), [P2\\_LIN\\_Msg\\_Write](#), [P2\\_LIN\\_Msg\\_Transmit](#)

**Example**

see [P2\\_LIN\\_Init](#)

**P2\_LIN\_Msg\_Write** configures a message box of a LIN interface for send or receive.

## Syntax

```
#Include ADwinPro_All.Inc
```

```
P2_LIN_Msg_Write(lin_datatable[], channel, membox,  
                msg_id, msg_dat[], msg_len, msg_send)
```

## Parameters

<b>lin_data-</b> <b>table[]</b>	Array with settings for data transfer between ADwin CPU and the LIN module.	ARRAY LONG
<b>channel</b>	Number (1...2) of LIN interface.	LONG
<b>membox</b>	Number (1...64) of the LIN message box to be configured.	LONG
<b>msg_id</b>	Identifier (0...63) of the message box.	LONG
<b>msg_dat[]</b>	Source array, from which data bytes are transferred in to the message box.	LONG
<b>msg_len</b>	Number (1...8) of data bytes of <b>msg_dat[]</b> to be transferred.	LONG
<b>msg_send</b>	Transfer status of the message box: 0: receive 1: send	LONG

## Notes

The array **msg\_dat[]** must be dimensioned to 8 elements at least. With message box transfer status "receive", the data of array **msg\_dat[]** will not be used.

After configuring, the message box is immediately active on the LIN bus, i.e. data can be received or sent.

If you want to change the data bytes for a message box with transfer status "send", use **P2\_LIN\_Msg\_Write** again.

The message box of a LIN master node operates different from a LIN slave node:

- **Master node, send:** The LIN master sends both the header (see **P2\_LIN\_Msg\_Transmit**) and then the data packet of the message box.
- **Master node, receive:** The LIN master sends the header (see **P2\_LIN\_Msg\_Transmit**) on the LIN Bus and waits for the response of the appropriate slave node. The received data packet is stored into the message box.
- **Slave node, send:** The LIN slave waits until the master sends the header with the identifier which fits to the identifier of the message box. Only then the slave node will its data packet.
- **Slave node, receive:** The slave node waits until the master sends the header with the identifier which fits to the identifier of the message box. Then the slave receives the data packet and stores it into the message box.

## Valid for

LIN-2 Rev. E

## P2\_LIN\_Msg\_Write

**See also**

[P2\\_LIN\\_Init](#), [P2\\_LIN\\_Init\\_Write](#), [P2\\_LIN\\_Init\\_Apply](#), [P2\\_LIN\\_Reset](#),  
[P2\\_LIN\\_Get\\_Version](#), [P2\\_LIN\\_Read\\_Dat](#), [P2\\_LIN\\_Msg\\_Transmit](#)

**Example**

see [P2\\_LIN\\_Init](#)



**P2\_LIN\_Msg\_Transmit** sends a header to the LIN bus. To use only with operating mode LIN master.

## Syntax

```
#Include ADwinPro_All.Inc

P2_LIN_Msg_Transmit(lin_datatable[], channel,
                    membox)
```

## Parameters

<b>lin_data-</b> <b>table[ ]</b>	Array with settings for data transfer between ADwin CPU and the LIN module.	ARRAY LONG
<b>channel</b>	Number (1...2) of LIN interface.	LONG
<b>membox</b>	Identifier (1...64) of the message box, which is enabled for data transfer.	LONG

## Notes

**P2\_LIN\_Msg\_Transmit** is valid only for an interface with operating mode LIN master (see **P2\_LIN\_Init\_Write**), because only a LIN master can send a header.

After sending a header to the LIN bus, only this bus node will react (which can also be the master node itself) which manages a message box with the identifier **membox**. Thus, this node will send a data packet to or receive a data packet from the LIN bus.

## Valid for

LIN-2 Rev. E

## See also

[P2\\_LIN\\_Init](#), [P2\\_LIN\\_Init\\_Write](#), [P2\\_LIN\\_Init\\_Apply](#), [P2\\_LIN\\_Reset](#), [P2\\_LIN\\_Get\\_Version](#), [P2\\_LIN\\_Read\\_Dat](#), [P2\\_LIN\\_Msg\\_Write](#)

## Example

see [P2\\_LIN\\_Init](#)

## P2\_LIN\_Msg\_Transmit

## P2\_LIN\_Set\_LED

**P2\_LIN\_Set\_LED** switches the additional LED of a LIN interface on (with color) or off.

### Syntax

```
#Include ADwinPro_All.Inc  
  
P2_LIN_Set_LED(module, channel, led_col)
```

### Parameters

module	Selected module address (1...15).	LONG
channel	Number (1...2) of LIN interface.	LONG
led_col	Status and color of the additional LED: 0: LED off. 1: LED on, color red. 2: LED on, color green. 3: LED on, color orange.	LONG

### Notes

- / -

### See also

P2\_Set\_LED

### Valid for

LIN-2 Rev. E

### Example

```
#Include ADwinPro_All.Inc  
Dim lin_datatable[150] As Long  
Dim ret_val As Long  
  
Init:  
    Rem initialize LIN controller  
    ret_val = P2_LIN_Init(1, lin_datatable)  
    P2_LIN_Set_LED(1, 1, 3)      'set LED 1 to orange
```

### 3.12 Pro II: Profibus interface

This section contains instructions to access a Profibus interface of *ADwin-Pro II*.

- [P2\\_Init\\_Profibus](#) (page 264)
- [P2\\_Run\\_Profibus](#) (page 266)

In the [Instruction List sorted by Module Types](#) (annex A.2) you will find which of the functions corresponds to the *ADwin-Pro II* modules.

## P2\_Init\_Profibus

**P2\_Init\_Profibus** initializes the Profibus Slave.

### Syntax

```
#Include ADwinGoldIII.Inc

ret_val = P2_Init_Profibus(module, dev_adr,
    in_mod_cnt, in_mod_type, out_mod_cnt,
    out_mod_type, work_arr[], info[])
```

### Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>dev_adr</b>	Slave node address (1...125) on the Profibus.	LONG
<b>in_mod_cnt</b>	Number (0...76) of input areas in the Profibus Slave. The max. number depends on <b>in_mod_type</b> .	LONG
<b>in_mod_type</b>	Key number (1...3, 16) for the length of input areas: 1: 1 Byte; max. value for <b>in_mod_cnt</b> : 76. 2: 2 Byte; max. value for <b>in_mod_cnt</b> : 38. 3: 4 Byte; max. value for <b>in_mod_cnt</b> : 19. 16:8 Byte; max. value for <b>in_mod_cnt</b> : 9.	LONG
<b>out_mod_cnt</b>	Number (0...76) of output areas in the Profibus Slave. The max. number depends on <b>out_mod_type</b> .	LONG
<b>out_mod_type</b>	Key number (1...3, 16) for the length of output areas: 1: 1 Byte; max. value for <b>out_mod_type</b> : 76. 2: 2 Byte; max. value for <b>out_mod_type</b> : 38. 3: 4 Byte; max. value for <b>out_mod_type</b> : 19. 16:8 Byte; max. value for <b>out_mod_type</b> : 9.	LONG
<b>work_arr[]</b>	Array to store data for operation of the Profibus Slave. The array must have at least 200 elements.	ARRAY LONG
<b>info[]</b>	Array holding data about the Profibus Slave. The array must have at least 10 elements. The elements <b>info[1]</b> and <b>info[2]</b> contain the production type of the Profibus Slave: <b>info[1]=1, info[2]=4</b>	ARRAY LONG
<b>ret_val</b>	State of initialization: 0: no error. ≠0: Error; please contact the support of Jäger Messtechnik.	LONG

### Notes

This instruction must be processed before working with Profibus Slave.

**P2\_Init\_Profibus** should be processed in a program section with low priority, because of the long processing time (about 2-3 seconds). Using the instruction in a (non-interruptable) high priority process, the communication between PC and ADwin system would be interrupted too long and thus produce an error message (timeout).

Station address, number and length of modules must equal the project settings of the profibus. For projecting, the module length is also given in words: 1 word = 2 byte.



## Valid for

Profi-SL Rev. E

## See also

[P2\\_Run\\_Profibus](#)

## Example

```
#Include ADwinPro_All.INC
#Define module 5          'module address
#Define node 2            'slave node address
#Define info Data_1       'info array
#Define out_arr Data_2
#Define in_arr Data_3

Dim out_arr[76] As Long At DM_Local
Dim in_arr[76] As Long At DM_Local
Dim conf_arr[200] As Long At DM_Local
Dim info[10] As Long At DM_Local
Dim i As Long
Dim error As Long

Init:
    Processdelay = 3000000    'set to 100 Hz
    For i = 1 To 10          'initialize info array
        info[i] = 0
    Next i
    Rem initialize profibus interface: 38 input data areas of 2 byte
    Rem and 76 output data bytes of 1 Byte
    error = P2_Init_Profibus(module,node,38,2,76,1,conf_arr,info)
    If (error <> 0) Then      'initialization error
        Par_1 = error
        Exit
    EndIf

Event:
    Rem set data in out_arr[] to be transferred
    For i = 1 To 76
        out_arr[i] = (out_arr[i] + i) And 0FFh
    Next i

    Rem send and read data (output areas: 76; input areas: 38)
    error = P2_Run_Profibus(module,out_arr,76,in_arr,38,conf_arr)
    error = error And 7h
    Par_2 = error

    Rem here the received data in in_arr[] can be processed
```

## P2\_Run\_Profibus

**P2\_Run\_Profibus** exchanges data with the Profibus Slave.

### Syntax

```
#Include ADwinGoldIII.Inc

ret_val = P2_Run_Profibus(module, out_pd_arr[],
    out_pd_arr_len, in_pd_arr[], in_pd_arr_len,
    work_arr[])
```

### Parameters

<code>module</code>	Selected module address (1...15).	LONG
<code>out_pd_arr[]</code>	Array from which the Profibus Slave reads data and writes them to the Profibus.	ARRAY LONG
<code>out_pd_arr_len</code>	Number of output areas (1...76), the data of which are read from the array <code>out_pd_arr[]</code> . The number may not be greater than given in <code>out_mod_cnt</code> with <b>P2_Init_Profibus</b> .	LONG
<code>in_pd_arr[]</code>	Array into which the Profibus-Slave writes data, which are read by the Profibus..	ARRAY LONG
<code>in_pd_arr_len</code>	Number of input areas (1...76), the data of which are returned in the array <code>in_pd_arr[]</code> . The number may not be greater than given in <code>in_mod_cnt</code> with <b>P2_Init_Profibus</b> .	LONG
<code>work_arr[]</code>	Array holding data for operation of the Profibus Slave, see <b>P2_Init_Profibus</b> .	ARRAY LONG
<code>ret_val</code>	Bit pattern holding the state of operation of the Profibus Slave. Only bits Bits 0...2 are important: <b>100b</b> : Slave is active and runs correctly. <b>010b</b> : Profibus inactive, Slave is waiting. <b>110b, 111b</b> : Error.	LONG

### Notes

**P2\_Run\_Profibus** should be processed in a program section with low priority, because of the long processing time. Using the instruction in a (non-interruptable) high priority process, the communication between PC and ADwin system would be interrupted too long and thus produce an error message (timeout).

Each array element in `in_pd_arr[]` and `out_pd_arr[]` contains a single data byte only (bits 0...7). Data areas of more than one byte length are saved in the appropriate number of consecutive array elements. Example: 5 data areas of 4 byte length are stored in 5×4=20 array elements.

### Valid for

Profi-SL Rev. E

### See also

[P2\\_Init\\_Profibus](#)

### Example

see [P2\\_Init\\_Profibus](#)

### 3.13 Pro II: EtherCAT interface

This section contains instructions to access a EtherCAT interface of *ADwin-Pro II*.

- [P2\\_ECAT\\_Get\\_Version](#) (page 268)
- [P2\\_ECAT\\_Get\\_State](#) (page 269)
- [P2\\_ECAT\\_Init](#) (page 270)
- [P2\\_ECAT\\_Read\\_Data\\_16L](#) (page 272)
- [P2\\_ECAT\\_Write\\_Data\\_16L](#) (page 273)

## P2\_ECAT\_Get\_Version

**P2\_ECAT\_Get\_Version** returns the version of the EtherCAT interface.

### Syntax

```
#Include ADwinPro_All.inc  
  
ret_val = P2_ECAT_Get_Version(ecat_datatable[])
```

### Parameters

<code>ecat_datatable[]</code>	Array which contains settings for data transfer between ADwin CPU and the EtherCAT module.	<b>ARRAY</b> <b>LONG</b>
<code>ret_val</code>	Version number of the EtherCAT interface, to be read in hexadecimal notation.	<b>LONG</b>

### Notes

The version number is only required if you have questions about programming the EtherCAT bus to our support.

The version number (in hexadecimal notation) has five digits, e.g. **10000h**; the first digits is the main revision number.

### Valid for

EtherCAT-SL Rev. E

### See also

[P2\\_ECAT\\_Init](#)

### Example

see [P2\\_ECAT\\_Init](#)



**P2\_ECAT\_Get\_State** returns the operation mode of the EtherCAT interface.

## Syntax

```
#Include ADwinPro_All.inc

ret_val = P2_ECAT_Get_State(ecat_datatable[])
```

## Parameters

<b>module</b>	Selected module address (1...15).	LONG
<b>ecat_datatable[]</b>	Array which contains settings for data transfer between <i>ADwin</i> CPU and the EtherCAT module.	ARRAY LONG
<b>ret_val</b>	Operation mode of the EtherCAT interface : 1: Operation mode Init. 2: Operation mode PreOp. 3: Operation mode Boot. 4: Operation mode SafeOp. 8: Operation mode Op.	LONG

## Notes

The operation mode Boot is not supported in *ADbasic*.

## Valid for

EtherCAT-SL Rev. E

## See also

[P2\\_ECAT\\_Init](#)

## Example

see [P2\\_ECAT\\_Init](#)

## P2\_ECAT\_Get\_State

## P2\_ECAT\_Init

**P2\_ECAT\_Init** initializes the EtherCAT Slave.

### Syntax

```
#Include ADwinPro_All.inc  
  
ret_val = P2_ECAT_Init(module, ecat_datatable[])
```

### Parameters

<code>module</code>	Selected module address (1...15).	LONG
<code>ecat_datatable[ ]</code>	Array which contains settings for data transfer between <i>ADwin</i> CPU and the EtherCAT module.	ARRAY LONG
<code>ret_val</code>	Status of initialization: 0: no error. 1: invalid module.	LONG

### Notes

This instruction must be run before using the EtherCAT Slave.

### Valid for

EtherCAT-SL Rev. E

### See also

[P2\\_ECAT\\_Get\\_Version](#)

## Example

```
#Include ADwinPro_All.INC

#Define ECAT_MODULE_ADDRESS 5
#Define ecat_inputs Data_1
#Define ecat_outputs Data_2

Dim ecat_inputs[16] As Long
Dim ecat_outputs[16] As Long
Dim ecat_comtable[150] As Long
Dim i As Long
Dim ret As Long

Init:
    Processdelay = 300000 ' 1kHz
    Rem initialize data transfer T11 <-> TiCo
    Par_1 = P2_ECAT_Init(ECAT_MODULE_ADDRESS, ecat_comtable)
    Par_2 = P2_ECAT_Get_Version(ecat_comtable) '10000h

    For i = 1 To 16
        ecat_inputs[i] = 0
    Next
    For i = 1 To 16
        ecat_outputs[i] = i
    Next
    Par_11 = 0
    Par_12 = 0

Event:
    ret = P2_ECAT_Get_State(ecat_comtable)

    If (ret = 8) Then 'operational mode
        ret = P2_ECAT_Write_Data_16L(ecat_comtable, ecat_outputs)
        If (ret = 0) Then 'writing data was o.k.
            Inc Par_11 'increase write counter
        EndIf

        ret = P2_ECAT_Read_Data_16L(ecat_comtable, ecat_inputs)
        If (ret = 0) Then 'reading data was o.k.
            Inc Par_12 'increase read counter
        EndIf
    EndIf
```

## P2\_ECAT\_Read\_Data\_16L

**P2\_ECAT\_Read\_Data\_16L** reads 16 Long values from the EtherCAT slave and returns them in an array.

### Syntax

```
#Include ADwinPro_All.inc  
  
ret_val = P2_ECAT_Read_Data_16L(ecat_datatable[],  
                                ecat_inputs[])
```

### Parameters

<code>ecat_datatable[]</code>	Array which contains settings for data transfer between ADwin CPU and the EtherCAT module.	<b>ARRAY</b> LONG
<code>ecat_inputs[]</code>	Array where the EtherCAT slave writes the Long data.	<b>ARRAY</b> LONG
<code>ret_val</code>	Status of reading: 0: Reading was successful. ≠0: Error while reading data.	LONG

### Notes

- / -

### Valid for

EtherCAT-SL Rev. E

### See also

[P2\\_ECAT\\_Init](#)

### Example

see [P2\\_ECAT\\_Init](#)

**P2\_ECAT\_Write\_Data\_16L** writes 16 Long values from an array to the EtherCAT slave.

## Syntax

```
#Include ADwinPro_All.inc

ret_val = P2_ECAT_Write_Data_16L(ecat_datatable[],
    ecat_outputs[])
```

## Parameters

<code>ecat_datatable[]</code>	Array which contains settings for data transfer between ADwin CPU and the EtherCAT module.	ARRAY LONG
<code>ecat_outputs[]</code>	Array from where the EtherCAT slave reads the Long data and writes them to the EtherCAT bus.	ARRAY LONG
<code>ret_val</code>	Status of writing: 0: Writing was successful. ≠0: Error while writing data.	LONG

## Notes

- / -

## Valid for

EtherCAT-SL Rev. E

## See also

[P2\\_ECAT\\_Init](#)

## Example

see [P2\\_ECAT\\_Init](#)

## P2\_ECAT\_Write\_Data\_16L

### 3.14 Pro II: FlexRay

This section contains instructions to access a FlexRay interface of *ADwin-Pro II*:

- [P2\\_FlexRay\\_Get\\_Version](#) (page 275)
- [P2\\_FlexRay\\_Init](#) (page 276)
- [P2\\_FlexRay\\_Read\\_Word](#) (page 278)
- [P2\\_FlexRay\\_Reset](#) (page 279)
- [P2\\_FlexRay\\_Set\\_LED](#) (page 280)
- [P2\\_FlexRay\\_Write\\_Word](#) (page 281)

In the [Instruction List sorted by Module Types](#) (annex A.2) you will find which of the functions corresponds to the *ADwin-Pro II* modules.

**P2\_FlexRay\_Get\_Version** returns the version number of the FlexRay interface.

## Syntax

```
#Include ADwinPro_All.inc

ret_val = P2_FlexRay_Get_Version(fr_datatable[],
                                status)
```

## Parameters

<b>fr_</b>	Array which contains settings for data transfer between ADwin CPU and the FlexRay module.	ARRAY
<b>datatable[ ]</b>		LONG
<b>status</b>	Status of access to the FlexRay module: 0: Access was successful. 1: Error: FlexRay controller was busy. 2: Error: FlexRay controller has not responded in time.	LONG
<b>ret_val</b>	Version number of FlexRay firmware.	LONG

## Notes

The version number will only be used, if you have questions about programming the FlexRay module to our support.

Each 4 hexadecimal numerals represent the version numbers of the high level and of the low level driver. For example **01030205h** represents the versions 1.3 (high level) and 2.5 (low level).

## See also

[P2\\_FlexRay\\_Init](#)

## Valid for

FlexRay-2 Rev. E

## Example

- / -

## P2\_FlexRay\_Get\_Version

## P2\_FlexRay\_Init

**P2\_FlexRay\_Init** initializes the data transfer between ADwin CPU and the FlexRay interface on a specified module.

### Syntax

```
#Include ADwinPro_All.inc

REM communication settings array of FlexRay module
Dim fr_datatable[150] As Long

P2_FlexRay_Init(module, fr_datatable[], status)
```

### Parameters

<code>module</code>	Selected module address (1...15).	LONG
<code>fr_datatable[ ]</code>	Array which contains settings for data transfer between ADwin CPU and the FlexRay module.	ARRAY LONG
<code>status</code>	Status of access to the FlexRay module: 0: Access was successful. 1: Error: No Pro II module at this address. 2: Error: no FlexRay interface on the module.	LONG

### Notes

**P2\_FlexRay\_Init** is to be executed before data transfer between ADwin CPU and FlexRay interface. The instruction should be used in the **Init:** section.

Before initialization, an array `fr_datatable[ ]` with 150 elements must be declared for each module.

### See also

[P2\\_FlexRay\\_Read\\_Word](#), [P2\\_FlexRay\\_Reset](#), [P2\\_FlexRay\\_Set\\_LED](#), [P2\\_FlexRay\\_Write\\_Word](#)

### Valid for

FlexRay-2 Rev. E



## Example

```
#Include ADwinPro_All.inc
Dim fr_datatable[150] As Long
Dim status, value As Long

Init:
    Rem initialize communication to the FlexRay controller
    P2_FlexRay_Init(1, fr_datatable, status)
    If (status <> 0) Then Exit

Event:
    Rem read address 210h from controller 1
    value = P2_FlexRay_Read_Word(fr_datatable,1,210h,status)
    If (status <> 0) Then End
    If (value = 15) Then
        Rem read address 220h from controller 1
        value = P2_FlexRay_Read_Word(fr_datatable,1,220h,status)
    Else
        Rem write value to address 192h of controller 1
        P2_FlexRay_Write_Word(fr_datatable,1,192h,value,status)
    EndIf

Finish:
    If (status <> 0) Then
        Rem set Par_1 to error number
        Par_1 = status
    EndIf
```

## P2\_FlexRay\_Read\_Word

**P2\_FlexRay\_Read\_Word** returns a 16 bit value from a FlexRay controller on the specified module.

### Syntax

```
#Include ADwinPro_All.inc  
  
ret_val = P2_FlexRay_Read_Word(fr_datatable[],  
                                controller, address, status)
```

### Parameters

<code>fr_datatable[]</code>	Array which contains settings for data transfer between ADwin CPU and the FlexRay module.	ARRAY LONG
<code>controller</code>	Number (1, 2) of the FlexRay controller.	LONG
<code>address</code>	Address (0...1FFEh) on the FlexRay controller, whose value is read. Enter the address with 2 byte alignment.	LONG
<code>status</code>	Status of access to the FlexRay module: 0: Access was successful. 1: Error: FlexRay controller was busy. 2: Error: FlexRay controller has not responded in time.	LONG
<code>ret_val</code>	Content (16 bit value) of the address on the FlexRay controller.	LONG

### Notes

- / -

### See also

[P2\\_FlexRay\\_Init](#), [P2\\_FlexRay\\_Reset](#), [P2\\_FlexRay\\_Write\\_Word](#)

### Valid for

FlexRay-2 Rev. E

### Example

see [P2\\_FlexRay\\_Init](#)

**P2\_FlexRay\_Reset** resets a FlexRay controller on the specified module.

## Syntax

```
#Include ADwinPro_All.inc
```

```
P2_FlexRay_Reset(fr_datatable[], controller, status)
```

## Parameters

<code>fr_datatable[]</code>	Array which contains settings for data transfer between ADwin CPU and the FlexRay module.	ARRAY LONG
<code>controller</code>	Number (1, 2) of the FlexRay controller.	LONG
<code>status</code>	Status of access to the FlexRay module: 0: Access was successful. 1: Error: FlexRay controller was busy. 2: Error: FlexRay controller has not responded in time.	LONG

## Notes

- / -

## See also

[P2\\_FlexRay\\_Init](#), [P2\\_FlexRay\\_Read\\_Word](#), [P2\\_FlexRay\\_Write\\_Word](#)

## Valid for

FlexRay-2 Rev. E

## Example

see [P2\\_FlexRay\\_Init](#)

## P2\_FlexRay\_Reset

## P2\_FlexRay\_Set\_LED

**P2\_FlexRay\_Set\_LED** switches a channel LED of a FlexRay controller on the specified module on or off.

### Syntax

```
#Include ADwinPro_All.inc

P2_FlexRay_Set_LED(module, controller, channel,
    value, status)
```

### Parameters

<b>fr_datatable</b>	Array which contains settings for data transfer between ADwin CPU and the FlexRay module.	ARRAY LONG
<b>controller</b>	Number (1, 2) of the FlexRay controller.	LONG
<b>channel</b>	Number (1, 2) of the FlexRay channel. 1: Channel A. 2: Channel B.	LONG
<b>value</b>	Status of the LED: 0: LED off. 1: LED on.	LONG
<b>status</b>	Status of access to the FlexRay module: 0: Access was successful. 1: Error: FlexRay controller was busy. 2: Error: FlexRay controller has not responded in time.	LONG

### Notes

- / -

### See also

[P2\\_FlexRay\\_Init](#)

### Valid for

FlexRay-2 Rev. E

### Example

```
#Include ADwinPro_All.inc
Dim fr_datatable[150] As Long
Dim status As Long

Init:
    Rem FlexRay-Controller initialisieren
    P2_FlexRay_Init(1, fr_datatable, status)
    Rem LED für Kanal 2, Controller 1 einschalten
    P2_FlexRay_Set_LED(fr_datatable,1,2,1,status)
```

**P2\_FlexRay\_Write\_Word** writes a 16 bit value to an address in a FlexRay controller of the specified module.

## Syntax

```
#Include ADwinPro_All.inc

P2_FlexRay_Write_Word(fr_datatable[],
    controller, address, value, status)
```

## Parameters

<b>fr_datatable[]</b>	Array which contains settings for data transfer between ADwin CPU and the FlexRay module.	ARRAY LONG
<b>controller</b>	Number (1, 2) of the FlexRay controller.	LONG
<b>address</b>	Address (0...1FFEh) on the FlexRay controller, where the value is written. Enter the address with 2 byte alignment.	LONG
<b>value</b>	16 bit value, which is writtten to the address in the FlexRay controller.	LONG
<b>status</b>	Status of access to the FlexRay module: 0: Access was successful. 1: Error: FlexRay controller was busy. 2: Error: FlexRay controller has not responded in time.	LONG

## Notes

- / -

## See also

[P2\\_FlexRay\\_Init](#), [P2\\_FlexRay\\_Read\\_Word](#), [P2\\_FlexRay\\_Reset](#)

## Valid for

FlexRay-2 Rev. E

## Example

see [P2\\_FlexRay\\_Init](#)

## P2\_FlexRay\_Write\_Word



## 4 Program Examples

### 4.1 Continuous signal conversion

This section describes the following instructions:

- Convert 1 channel (Seite 283)

The modules Pro II Aln-F-4/14 Rev. E and Pro II Aln-F-8/14 Rev. E allow for very fast, continuous signal conversion. In parallel to conversion, the data must also be read and if need be processed.

Hereafter there are examples for continuous signal conversion.

- Continuous signal conversion: Convert 1 channel, Seite 283

Most examples are stored as program files in the directory <C:\ADwin\ADbasic\samples\_ADwin\_PRO>.

#### Convert 1 channel

You need

- one module Pro II Aln-F-x/14 Rev. E with module address 1.
- an analog signal at input channel 1.

The program <PROII-F-x-14-CONT-1CH.BAS> does a continuous burst sequence on input channel 1 with a clock rate 25MHz. The memory is set to hold 20000 measurement values.

During the running burst sequence the measurement values are read into the global array `Data_1` (a FIFO array is not available). The parallel conversion and read-out calls for an adjustment to each other. For this the memory area is divided into 4 ranges of 5000 values. That range will only be read, which has just been written completely.

In the same way, an adjustment of conversion rate and read-out rate is necessary. The process cycle is set by `Processdelay = 20000` (= 15kHz) in a way, so the read-out rate in average is a multiple of the conversion rate 25MHz:

$$15\text{kHz} \cdot 5000 = \text{Read-out rate } 75\text{ kHz} > \text{Conversion rate } 25\text{MHz}$$

For changes of the example please note: If the processing time of the **Event:** section rises, e.g. by processing measurement values, the read-out rate may be too low to read all converted values. In this case measurement values will be lost, because they are overwritten. Thus, you have to adjust conversion rate and read-out rate anew.



```
#Include ADwinPro_All.Inc 'include file
#Define module 1 'module no.
#Define d1 Data_1 'holds values of channel 1
#Define mem_idx Par_1 'mem position of last written value
#Define max_val 20000 'no. of values
#Define seg1 max_val/8 'end of segment 1
#Define seg2 max_val/4 'end of segment 2
#Define seg3 max_val/8*3 'end of segment 3
#Define blk max_val/4 'read block size

Dim d1[max_val] As Long 'destination array
Dim pattern As Long 'bit pattern to address one module
Dim segment As Long 'segment that is currently written

Init:
    REM 1 channel continuous, mem for max_val values, 25 MHz
    P2_Burst_Init(module,1,0,max_val,2,010b)
    pattern = Shift_Left(1,module-1)'address this module only
    P2_Burst_Start(pattern)
    segment = 1 'start with memory segment 1
    Processdelay = 20000 'cycle time 66.6 µs -> 15 kHz

Event:
    mem_idx = P2_Burst_Read_Index(module) 'get current mem index
    If (segment = 1) Then 'read 1. segment
        If ((mem_idx > seg1) And (mem_idx < seg3)) Then
            REM memory index is in segments 2 or 3: read segment 1
            P2_Burst_Read_Unpacked1(module,blk,0,Data_1,1,3)
            segment = 2
        EndIf
    EndIf

    If (segment = 2) Then 'read 2. segment
        If (mem_idx > seg2) Then
            REM memory index is in segments 3 or 4: read segment 2
            P2_Burst_Read_Unpacked1(module,blk,seg1,Data_1,blk+1,3)
            segment = 3
        EndIf
    EndIf

    If (segment = 3) Then 'read 3. segment
        If ((mem_idx > seg3) Or (mem_idx < seg1)) Then
            REM memory index is in segments 4 or 1: read segment 3
            P2_Burst_Read_Unpacked1(module,blk,seg2,Data_1,blk*2+1,3)
            segment = 4
        EndIf
    EndIf

    If (segment = 4) Then 'read 4. segment
        If (mem_idx < seg2) Then
            REM memory index is in segments 1 or 2: read segment 4
            P2_Burst_Read_Unpacked1(module,blk,seg3,Data_1,blk*3+1,3)
            segment = 1
        EndIf
    EndIf
```



### Instruction Lists

#### A.1 Alphabetic Instruction List

##### A

P2\_ADC · 35  
P2\_ADC24 · 36  
P2\_ADCF · 89  
P2\_ADCF24 · 90  
P2\_ADCF\_Mode · 91  
P2\_ADCF\_Read\_Limit · 94  
P2\_ADCF\_Read\_Min\_Max4 · 97  
P2\_ADCF\_Read\_Min\_Max8 · 99  
P2\_ADCF\_Reset\_Min\_Max · 96  
P2\_ADCF\_Set\_Limit · 95  
P2\_ADC\_Read\_Limit · 37  
P2\_ADC\_Set\_Limit · 39

##### B

P2\_Burst\_CRead\_Unpacked1 · 58  
P2\_Burst\_CRead\_Unpacked2 · 60  
P2\_Burst\_CRead\_Unpacked4 · 62  
P2\_Burst\_CRead\_Unpacked8 · 64  
P2\_Burst\_Init · 66  
P2\_Burst\_Read · 71  
P2\_Burst\_Read\_Index · 69  
P2\_Burst\_Read\_Unpacked1 · 73  
P2\_Burst\_Read\_Unpacked2 · 75  
P2\_Burst\_Read\_Unpacked4 · 77  
P2\_Burst\_Read\_Unpacked8 · 79  
P2\_Burst\_Reset · 81  
P2\_Burst\_Start · 83  
P2\_Burst\_Status · 84  
P2\_Burst\_Stop · 86

##### C

P2\_CAN\_Interrupt\_Source · 220  
CAN\_Msg · 218  
P2\_Check\_LED · 5  
P2\_Cnt\_Clear · 162  
P2\_Cnt\_Enable · 163  
P2\_Cnt\_Get\_PW · 165  
P2\_Cnt\_Get\_PW\_HL · 166  
P2\_Cnt\_Get\_Status · 164  
P2\_Cnt\_Latch · 167  
P2\_Cnt\_Mode · 168  
P2\_Cnt\_PW\_Enable · 170  
P2\_Cnt\_PW\_Latch · 171  
P2\_Cnt\_Read · 172  
P2\_Cnt\_Read4 · 173  
P2\_Cnt\_Read\_Int\_Register · 174  
P2\_Cnt\_Read\_Latch · 175  
P2\_Cnt\_Read\_Latch4 · 176  
P2\_Cnt\_Sync\_Latch · 177  
CPU\_Digin (T11) · 12  
CPU\_Digout · 13  
CPU\_Dig\_IO\_Config · 14  
CPU\_Event\_Config · 15

##### D

P2\_DAC · 117  
P2\_DAC4 · 118  
P2\_DAC4\_Packed · 119  
P2\_DAC8 · 121  
P2\_DAC8\_Packed · 122  
P2\_Digin\_Edge · 136  
P2\_Digin\_FIFO\_Clear · 137  
P2\_Digin\_FIFO\_Enable · 138  
P2\_Digin\_FIFO\_Full · 140  
P2\_Digin\_FIFO\_Read · 141  
P2\_Digin\_Fifo\_Read\_Fast · 143  
P2\_Digin\_FIFO\_Read\_Timer · 145  
P2\_Digin\_Long · 146  
P2\_Digout · 147  
P2\_Digout\_Bits · 148  
P2\_Digout\_FIFO\_Clear · 149  
P2\_Digout\_FIFO\_Empty · 150  
P2\_Digout\_FIFO\_Enable · 151  
P2\_Digout\_FIFO\_Read\_Timer · 152  
P2\_Digout\_FIFO\_Start · 153  
P2\_Digout\_FIFO\_Write · 154  
P2\_Digout\_Long · 156  
P2\_Digout\_Reset · 157  
P2\_Digout\_Set · 158  
P2\_Digprog · 159  
P2\_Dig\_FIFO\_Mode · 132  
P2\_Dig\_Latch · 133  
P2\_Dig\_Read\_Latch · 134  
P2\_Dig\_Write\_Latch · 135

##### E

P2\_Event2\_Config · 9  
P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11

##### F

P2\_FlexRay\_Get\_Version · 275  
P2\_FlexRay\_Init · 276  
P2\_FlexRay\_Read\_Word · 278  
P2\_FlexRay\_Reset · 279  
P2\_FlexRay\_Set\_LED · 280  
P2\_FlexRay\_Write\_Word · 281

##### G

P2\_Get\_Digout\_Long · 160  
P2\_MIO\_Get\_Digout\_Long · 32

##### L

P2\_LIN\_Get\_Version · 256  
P2\_LIN\_Init · 251  
P2\_LIN\_Init\_Apply · 254  
P2\_LIN\_Init\_Write · 253

P2\_LIN\_Msg\_Transmit · 261  
P2\_LIN\_Msg\_Write · 259  
P2\_LIN\_Read\_Dat · 257  
P2\_LIN\_Reset · 255  
P2\_LIN\_Set\_LED · 262

## M

P2\_MIO\_Digin\_Long · 28  
P2\_MIO\_Digout · 29  
P2\_MIO\_Digout\_Long · 30  
P2\_MIO\_DigProg · 31  
P2\_MIO\_Dig\_Latch · 25  
P2\_MIO\_Dig\_Read\_Latch · 26  
P2\_MIO\_Dig\_Write\_Latch · 27

## P

P2\_CAN\_Set\_LED · 222  
P2\_Check\_Shift\_Reg · 240  
P2\_ECAT\_Get\_State · 269  
P2\_ECAT\_Get\_Version · 268  
P2\_ECAT\_Init · 270  
P2\_ECAT\_Read\_Data\_16L · 272  
P2\_ECAT\_Write\_Data\_16L · 273  
P2\_En\_Interrupt · 223  
P2\_En\_Receive · 225  
P2\_En\_Transmit · 226  
P2\_Get\_CAN\_Reg · 227  
P2\_Get\_RS · 241  
P2\_Init\_CAN · 228  
P2\_Init\_Profibus · 264  
P2\_PWM\_Latch · 193  
P2\_PWM\_Reset · 194  
P2\_PWM\_Standby\_Value · 195  
P2\_PWM\_Write\_Latch · 196  
P2\_PWM\_Write\_Latch\_Block · 197  
P2\_Read\_FIFO · 242  
P2\_Read\_Msg · 229  
P2\_Read\_Msg\_Con · 231  
P2\_RS485\_Send · 246  
P2\_RS\_Init · 243  
P2\_RS\_Reset · 245  
P2\_Run\_Profibus · 266  
P2\_Seq\_Init · 45  
P2\_Seq\_Read · 48  
P2\_Seq\_Read24 · 49  
P2\_Seq\_Read\_Packed · 51  
P2\_Seq\_Start · 52  
P2\_Seq\_Wait · 53  
P2\_Set\_CAN\_Baudrate · 233  
P2\_Set\_CAN\_Reg · 237  
P2\_Set\_Mux · 54  
P2\_Set\_RS · 248  
P2\_TC\_Latch · 209  
P2\_TC\_Read\_Latch · 210  
P2\_TC\_Read\_Latch4 · 212  
P2\_TC\_Read\_Latch8 · 214  
P2\_TC\_Set\_Rate · 216  
P2\_Transmit · 238  
P2\_Write\_Fifo · 249

P2\_PWM\_Enable · 189  
P2\_PWM\_Get\_Status · 190  
P2\_PWM\_Init · 191  
P2\_PWM\_Latch · 193  
P2\_PWM\_Reset · 194  
P2\_PWM\_Standby\_Value · 195  
P2\_PWM\_Write\_Latch · 196  
P2\_PWM\_Write\_Latch\_Block · 197

## R

P2\_Read\_ADC · 40  
P2\_Read\_ADC24 · 41  
P2\_Read\_ADCF · 101  
P2\_Read\_ADCF32 · 109  
P2\_Read\_ADCF4 · 103  
P2\_Read\_ADCF4\_24B · 104  
P2\_Read\_ADCF4\_Packed · 107  
P2\_Read\_ADCF8 · 105  
P2\_Read\_ADCF8\_24B · 106  
P2\_Read\_ADCF8\_Packed · 108  
P2\_Read\_ADCF\_24 · 102  
P2\_Read\_ADCF\_SConv · 110  
P2\_Read\_ADCF\_SConv24 · 111  
P2\_Read\_ADCF\_SConv32 · 112  
P2\_Read\_ADC\_SConv · 42  
P2\_Read\_ADC\_SConv24 · 43  
P2\_RS\_Set\_LED · 247  
P2\_RTD\_Channel\_Config · 200  
P2\_RTD\_Config · 202  
P2\_RTD\_Convert · 203  
P2\_RTD\_Read · 204  
P2\_RTD\_Read8 · 205  
P2\_RTD\_Start · 206  
P2\_RTD\_Status · 208

## S

P2\_Set\_Average\_Filter · 88  
P2\_Set\_Gain · 113  
P2\_Set\_LED · 6  
P2\_SE\_Diff · 44  
P2\_SSI\_Mode · 179  
P2\_SSI\_Read · 180  
P2\_SSI\_Read2 · 182  
P2\_SSI\_Set\_Bits · 183  
P2\_SSI\_Set\_Clock · 184  
P2\_SSI\_Set\_Delay · 185  
P2\_SSI\_Start · 186  
P2\_SSI\_Status · 187  
P2\_Start\_Conv · 55  
P2\_Start\_ConvF · 114  
P2\_Start\_DAC · 124  
P2\_Sync\_All · 16  
P2\_Sync\_Enable · 18  
P2\_Sync\_Mode · 20  
P2\_Sync\_Stat · 22

## W

P2\_Wait\_EOC · 56

P2\_Wait\_EOCF · 115  
P2\_Wait\_Mux · 57  
P2\_Write\_DAC · 125  
P2\_Write\_DAC32 · 130  
P2\_Write\_DAC4 · 126  
P2\_Write\_DAC4\_Packed · 127  
P2\_Write\_DAC8 · 128  
P2\_Write\_DAC8\_Packed · 129

## A.2 Instruction List sorted by Module Types

You find the instruction lists of the modules on these pages:

Modulname	Seite
Aln-16/18-8B Rev. E	A-4
Aln-32/18 Rev. E	A-4
Aln-8/18 Rev. E	A-5
Aln-8/18-8B Rev. E	A-5
Aln-F-4/14 Rev. E	A-5
Aln-F-4/16 Rev. E	A-6
Aln-F-4/18 Rev. E	A-6
Aln-F-8/14 Rev. E	A-7
Aln-F-8/16 Rev. E	A-7
Aln-F-8/18 Rev. E	A-8
AOut-4/16 Rev. E	A-8
AOut-8/16 Rev. E	A-8
CAN-2 Rev. E	A-8
CNT-D Rev. E	A-9
CNT-I Rev. E	A-9
CNT-T Rev. E	A-9
CPU-T11	A-9
DIO-32 Rev. E	A-9
DIO-32-TiCo Rev. E	A-10
EtherCAT-SL Rev. E	A-10
FlexRay-2 Rev. E	A-10
LIN-2 Rev. E	A-10
MIO-4 Rev. E	A-10
MIO-4-ET1 Rev. E	A-11
OPT-16 Rev. E	A-11
OPT-32 Rev. E	A-11
Profi-SL Rev. E	A-11
Profi-SL Rev. E Rev. E	A-11
PWM-16(-I) Rev. E	A-12
REL-16 Rev. E	A-12
RSxxx-2 Rev. E	A-12
RSxxx-4 Rev. E	A-12
RTD-8 Rev. E	A-12
TC-8-ISO Rev. A	A-12
TC-8-ISO Rev. E	A-12
TRA-16 Rev. E	A-12

### Aln-16/18-8B Rev. E

- A:** P2\_ADC · 35  
P2\_ADC24 · 36  
P2\_ADC\_Read\_Limit · 37  
P2\_ADC\_Set\_Limit · 39
- C:** P2\_Check\_LED · 5
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- R:** P2\_ADC\_SConv · 42  
P2\_ADC\_SConv24 · 43  
P2\_Read\_ADC · 40  
P2\_Read\_ADC24 · 41
- S:** P2\_Seq\_Init · 45  
P2\_Seq\_Read · 48  
P2\_Seq\_Read24 · 49  
P2\_Seq\_Read\_Packed · 51  
P2\_Seq\_Start · 52  
P2\_Seq\_Wait · 53  
P2\_Set\_LED · 6  
P2\_Set\_Mux · 54  
P2\_SE\_Diff · 44  
P2\_Start\_Conv · 55  
P2\_Sync\_All · 16
- W:** P2\_Wait\_EOC · 56  
P2\_Wait\_Mux · 57

### Aln-32/18 Rev. E

- A:** P2\_ADC · 35  
P2\_ADC24 · 36  
P2\_ADC\_Read\_Limit · 37  
P2\_ADC\_Set\_Limit · 39
- C:** P2\_Check\_LED · 5
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- R:** P2\_Read\_ADC · 40  
P2\_Read\_ADC24 · 41  
P2\_Read\_ADC\_SConv · 42  
P2\_Read\_ADC\_SConv24 · 43
- S:** P2\_Seq\_Init · 45  
P2\_Seq\_Read · 48  
P2\_Seq\_Read24 · 49  
P2\_Seq\_Read\_Packed · 51  
P2\_Seq\_Start · 52  
P2\_Seq\_Wait · 53  
P2\_Set\_LED · 6  
P2\_Set\_Mux · 54  
P2\_SE\_Diff · 44  
P2\_Start\_Conv · 55  
P2\_Sync\_All · 16
- W:** P2\_Wait\_EOC · 56  
P2\_Wait\_Mux · 57

## **Aln-8/18 Rev. E**

- A:** [P2\\_ADC · 35](#)  
[P2\\_ADC24 · 36](#)  
[P2\\_ADC\\_Read\\_Limit · 37](#)  
[P2\\_ADC\\_Set\\_Limit · 39](#)
- C:** [P2\\_Check\\_LED · 5](#)
- E:** [P2\\_Event\\_Config · 8](#)  
[P2\\_Event\\_Enable · 7](#)  
[P2\\_Event\\_Read · 11](#)
- R:** [P2\\_Read\\_ADC · 40](#)  
[P2\\_Read\\_ADC24 · 41](#)  
[P2\\_Read\\_ADC\\_SConv · 42](#)  
[P2\\_Read\\_ADC\\_SConv24 · 43](#)
- S:** [P2\\_Seq\\_Init · 45](#)  
[P2\\_Seq\\_Read · 48](#)  
[P2\\_Seq\\_Read24 · 49](#)  
[P2\\_Seq\\_Read\\_Packed · 51](#)  
[P2\\_Seq\\_Start · 52](#)  
[P2\\_Seq\\_Wait · 53](#)  
[P2\\_Set\\_LED · 6](#)  
[P2\\_Set\\_Mux · 54](#)  
[P2\\_Start\\_Conv · 55](#)  
[P2\\_Sync\\_All · 16](#)
- W:** [P2\\_Wait\\_EOC · 56](#)  
[P2\\_Wait\\_Mux · 57](#)

## **Aln-8/18-8B Rev. E**

- A:** [P2\\_ADC · 35](#)  
[P2\\_ADC24 · 36](#)  
[P2\\_ADC\\_Read\\_Limit · 37](#)  
[P2\\_ADC\\_Set\\_Limit · 39](#)
- C:** [P2\\_Check\\_LED · 5](#)
- E:** [P2\\_Event\\_Config · 8](#)  
[P2\\_Event\\_Enable · 7](#)  
[P2\\_Event\\_Read · 11](#)
- R:** [P2\\_ADC\\_SConv · 42](#)  
[P2\\_ADC\\_SConv24 · 43](#)  
[P2\\_Read\\_ADC · 40](#)  
[P2\\_Read\\_ADC24 · 41](#)
- S:** [P2\\_Seq\\_Init · 45](#)  
[P2\\_Seq\\_Read · 48](#)  
[P2\\_Seq\\_Read24 · 49](#)  
[P2\\_Seq\\_Read\\_Packed · 51](#)  
[P2\\_Seq\\_Start · 52](#)  
[P2\\_Seq\\_Wait · 53](#)  
[P2\\_Set\\_LED · 6](#)  
[P2\\_Set\\_Mux · 54](#)  
[P2\\_Start\\_Conv · 55](#)  
[P2\\_Sync\\_All · 16](#)
- W:** [P2\\_Wait\\_EOC · 56](#)  
[P2\\_Wait\\_Mux · 57](#)

## **Aln-F-4/14 Rev. E**

- A:** [P2\\_ADCF · 89](#)  
[P2\\_ADCF\\_Read\\_Limit · 94](#)  
[P2\\_ADCF\\_Set\\_Limit · 95](#)
- B:** [P2\\_Burst\\_CRead\\_Unpacked1 · 58](#)  
[P2\\_Burst\\_CRead\\_Unpacked2 · 60](#)  
[P2\\_Burst\\_CRead\\_Unpacked4 · 62](#)  
[P2\\_Burst\\_Init · 66](#)  
[P2\\_Burst\\_Read · 71](#)  
[P2\\_Burst\\_Read\\_Index · 69](#)  
[P2\\_Burst\\_Read\\_Unpacked1 · 73](#)  
[P2\\_Burst\\_Read\\_Unpacked2 · 75](#)  
[P2\\_Burst\\_Read\\_Unpacked4 · 77](#)  
[P2\\_Burst\\_Reset · 81](#)  
[P2\\_Burst\\_Start · 83](#)  
[P2\\_Burst\\_Status · 84](#)  
[P2\\_Burst\\_Stop · 86](#)
- C:** [P2\\_Check\\_LED · 5](#)
- E:** [P2\\_Event2\\_Config · 9](#)  
[P2\\_Event\\_Config · 8](#)  
[P2\\_Event\\_Enable · 7](#)  
[P2\\_Event\\_Read · 11](#)
- R:** [P2\\_Read\\_ADCF · 101](#)  
[P2\\_Read\\_ADCF32 · 109](#)  
[P2\\_Read\\_ADCF4 · 103](#)  
[P2\\_Read\\_ADCF4\\_Packed · 107](#)
- S:** [P2\\_Set\\_Average\\_Filter · 88](#)  
[P2\\_Set\\_LED · 6](#)  
[P2\\_Sync\\_All · 16](#)  
[P2\\_Sync\\_Enable · 18](#)  
[P2\\_Sync\\_Mode · 20](#)  
[P2\\_Sync\\_Stat · 22](#)

**Aln-F-4/16 Rev. E**

- A:** P2\_ADCF · 89  
P2\_ADCF\_Mode · 91  
P2\_ADCF\_Read\_Limit · 94  
P2\_ADCF\_Read\_Min\_Max4 · 97  
P2\_ADCF\_Read\_Min\_Max8 · 99  
P2\_ADCF\_Reset\_Min\_Max · 96  
P2\_ADCF\_Set\_Limit · 95
- B:** P2\_Burst\_CRead\_Unpacked1 · 58  
P2\_Burst\_CRead\_Unpacked2 · 60  
P2\_Burst\_CRead\_Unpacked4 · 62  
P2\_Burst\_Init · 66  
P2\_Burst\_Read · 71  
P2\_Burst\_Read\_Index · 69  
P2\_Burst\_Read\_Unpacked1 · 73  
P2\_Burst\_Read\_Unpacked2 · 75  
P2\_Burst\_Read\_Unpacked4 · 77  
P2\_Burst\_Reset · 81  
P2\_Burst\_Start · 83  
P2\_Burst\_Status · 84  
P2\_Burst\_Stop · 86
- C:** P2\_Check\_LED · 5
- E:** P2\_Event2\_Config · 9  
P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- R:** P2\_Read\_ADCF · 101  
P2\_Read\_ADCF32 · 109  
P2\_Read\_ADCF4 · 103  
P2\_Read\_ADCF4\_Packed · 107  
P2\_Read\_ADCF\_SConv · 110  
P2\_Read\_ADCF\_SConv32 · 112
- S:** P2\_Set\_Average\_Filter · 88  
P2\_Set\_LED · 6  
P2\_Start\_ConvF · 114  
P2\_Sync\_All · 16  
P2\_Sync\_Enable · 18  
P2\_Sync\_Mode · 20  
P2\_Sync\_Stat · 22
- W:** P2\_Wait\_EOCF · 115

**Aln-F-4/18 Rev. E**

- A:** P2\_ADCF · 89  
P2\_ADCF24 · 90  
P2\_ADCF\_Mode · 91  
P2\_ADCF\_Read\_Limit · 94  
P2\_ADCF\_Set\_Limit · 95
- C:** P2\_Check\_LED · 5
- E:** P2\_Event2\_Config · 9  
P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- R:** P2\_Read\_ADCF · 101  
P2\_Read\_ADCF24 · 102  
P2\_Read\_ADCF32 · 109  
P2\_Read\_ADCF4 · 103  
P2\_Read\_ADCF4\_24B · 104  
P2\_Read\_ADCF4\_Packed · 107  
P2\_Read\_ADCF\_SConv · 110  
P2\_Read\_ADCF\_SConv24 · 111  
P2\_Read\_ADCF\_SConv32 · 112
- S:** P2\_Set\_LED · 6  
P2\_Start\_ConvF · 114  
P2\_Sync\_All · 16  
P2\_Sync\_Enable · 18  
P2\_Sync\_Mode · 20  
P2\_Sync\_Stat · 22
- W:** P2\_Wait\_EOCF · 115

## **Aln-F-8/14 Rev. E**

- A:** P2\_ADCF · 89  
P2\_ADCF\_Read\_Limit · 94  
P2\_ADCF\_Set\_Limit · 95
- B:** P2\_Burst\_CRead\_Unpacked1 · 58  
P2\_Burst\_CRead\_Unpacked2 · 60  
P2\_Burst\_CRead\_Unpacked4 · 62  
P2\_Burst\_CRead\_Unpacked8 · 64  
P2\_Burst\_Init · 66  
P2\_Burst\_Read · 71  
P2\_Burst\_Read\_Index · 69  
P2\_Burst\_Read\_Unpacked1 · 73  
P2\_Burst\_Read\_Unpacked2 · 75  
P2\_Burst\_Read\_Unpacked4 · 77  
P2\_Burst\_Read\_Unpacked8 · 79  
P2\_Burst\_Reset · 81  
P2\_Burst\_Start · 83  
P2\_Burst\_Status · 84  
P2\_Burst\_Stop · 86
- C:** P2\_Check\_LED · 5
- E:** P2\_Event2\_Config · 9  
P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- R:** P2\_Read\_ADCF · 101  
P2\_Read\_ADCF32 · 109  
P2\_Read\_ADCF4 · 103  
P2\_Read\_ADCF4\_Packed · 107  
P2\_Read\_ADCF8 · 105  
P2\_Read\_ADCF8\_Packed · 108
- S:** P2\_Set\_Average\_Filter · 88  
P2\_Set\_LED · 6  
P2\_Sync\_All · 16  
P2\_Sync\_Enable · 18  
P2\_Sync\_Mode · 20  
P2\_Sync\_Stat · 22

## **Aln-F-8/16 Rev. E**

- A:** P2\_ADCF · 89  
P2\_ADCF\_Mode · 91  
P2\_ADCF\_Read\_Limit · 94  
P2\_ADCF\_Read\_Min\_Max4 · 97  
P2\_ADCF\_Read\_Min\_Max8 · 99  
P2\_ADCF\_Reset\_Min\_Max · 96  
P2\_ADCF\_Set\_Limit · 95
- B:** P2\_Burst\_CRead\_Unpacked1 · 58  
P2\_Burst\_CRead\_Unpacked2 · 60  
P2\_Burst\_CRead\_Unpacked4 · 62  
P2\_Burst\_CRead\_Unpacked8 · 64  
P2\_Burst\_Init · 66  
P2\_Burst\_Read · 71  
P2\_Burst\_Read\_Index · 69  
P2\_Burst\_Read\_Unpacked1 · 73  
P2\_Burst\_Read\_Unpacked2 · 75  
P2\_Burst\_Read\_Unpacked4 · 77  
P2\_Burst\_Read\_Unpacked8 · 79  
P2\_Burst\_Reset · 81  
P2\_Burst\_Start · 83  
P2\_Burst\_Status · 84  
P2\_Burst\_Stop · 86
- C:** P2\_Check\_LED · 5
- E:** P2\_Event2\_Config · 9  
P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- R:** P2\_Read\_ADCF · 101  
P2\_Read\_ADCF32 · 109  
P2\_Read\_ADCF4 · 103  
P2\_Read\_ADCF4\_Packed · 107  
P2\_Read\_ADCF8 · 105  
P2\_Read\_ADCF8\_Packed · 108  
P2\_Read\_ADCF\_SConv · 110  
P2\_Read\_ADCF\_SConv32 · 112
- S:** P2\_Set\_Average\_Filter · 88  
P2\_Set\_LED · 6  
P2\_Start\_ConvF · 114  
P2\_Sync\_All · 16  
P2\_Sync\_Enable · 18  
P2\_Sync\_Mode · 20  
P2\_Sync\_Stat · 22
- W:** P2\_Wait\_EOCF · 115

**Aln-F-8/18 Rev. E**

- A:** P2\_ADCF · 89  
P2\_ADCF24 · 90  
P2\_ADCF\_Mode · 91  
P2\_ADCF\_Read\_Limit · 94  
P2\_ADCF\_Set\_Limit · 95
- C:** P2\_Check\_LED · 5
- E:** P2\_Event2\_Config · 9  
P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- R:** P2\_Read\_ADCF · 101  
P2\_Read\_ADCF32 · 109  
P2\_Read\_ADCF4 · 103  
P2\_Read\_ADCF4\_24B · 104  
P2\_Read\_ADCF4\_Packed · 107  
P2\_Read\_ADCF8 · 105  
P2\_Read\_ADCF8\_24B · 106  
P2\_Read\_ADCF8\_Packed · 108  
P2\_Read\_ADCF\_24 · 102  
P2\_Read\_ADCF\_SConv · 110  
P2\_Read\_ADCF\_SConv24 · 111  
P2\_Read\_ADCF\_SConv32 · 112
- S:** P2\_Set\_LED · 6  
P2\_Start\_ConvF · 114  
P2\_Sync\_All · 16  
P2\_Sync\_Enable · 18  
P2\_Sync\_Mode · 20  
P2\_Sync\_Stat · 22
- W:** P2\_Wait\_EOCF · 115

**AOut-4/16 Rev. E**

- C:** P2\_Check\_LED · 5
- D:** P2\_DAC · 117  
P2\_DAC4 · 118  
P2\_DAC4\_Packed · 119
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- S:** P2\_Set\_LED · 6  
P2\_Start\_DAC · 124  
P2\_Sync\_All · 16  
P2\_Sync\_Enable · 18  
P2\_Sync\_Stat · 22
- W:** P2\_Write\_DAC · 125  
P2\_Write\_DAC32 · 130  
P2\_Write\_DAC4 · 126  
P2\_Write\_DAC4\_Packed · 127

**AOut-8/16 Rev. E**

- C:** P2\_Check\_LED · 5
- D:** P2\_DAC · 117  
P2\_DAC4 · 118  
P2\_DAC4\_Packed · 119  
P2\_DAC8 · 121  
P2\_DAC8\_Packed · 122
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- S:** P2\_Set\_LED · 6  
P2\_Start\_DAC · 124  
P2\_Sync\_All · 16  
P2\_Sync\_Enable · 18  
P2\_Sync\_Stat · 22
- W:** P2\_Write\_DAC · 125  
P2\_Write\_DAC32 · 130  
P2\_Write\_DAC4 · 126  
P2\_Write\_DAC4\_Packed · 127  
P2\_Write\_DAC8 · 128  
P2\_Write\_DAC8\_Packed · 129

**CAN-2 Rev. E**

- C:** CAN\_Msg · 218  
P2\_CAN\_Set\_LED · 222  
P2\_Check\_LED · 5
- E:** P2\_CAN\_Interrupt\_Source · 220  
P2\_En\_Interrupt · 223  
P2\_En\_Receive · 225  
P2\_En\_Transmit · 226  
P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- G:** P2\_Get\_CAN\_Reg · 227
- I:** P2\_Init\_CAN · 228
- R:** P2\_Read\_Msg · 229  
P2\_Read\_Msg\_Con · 231
- S:** P2\_Set\_CAN\_Baudrate · 233  
P2\_Set\_CAN\_Reg · 237  
P2\_Set\_LED · 6
- T:** P2\_Transmit · 238



## CNT-D Rev. E

- C:** P2\_Check\_LED · 5  
P2\_Cnt\_Clear · 162  
P2\_Cnt\_Enable · 163  
P2\_Cnt\_Get\_PW · 165  
P2\_Cnt\_Get\_PW\_HL · 166  
P2\_Cnt\_Get\_Status · 164  
P2\_Cnt\_Latch · 167  
P2\_Cnt\_Mode · 168  
P2\_Cnt\_PW\_Enable · 170  
P2\_Cnt\_PW\_Latch · 171  
P2\_Cnt\_Read · 172  
P2\_Cnt\_Read4 · 173  
P2\_Cnt\_Read\_Int\_Register · 174  
P2\_Cnt\_Read\_Latch · 175  
P2\_Cnt\_Read\_Latch4 · 176  
P2\_Cnt\_Sync\_Latch · 177
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- S:** P2\_Set\_LED · 6  
P2\_SSI\_Mode · 179  
P2\_SSI\_Read · 180  
P2\_SSI\_Read2 · 182  
P2\_SSI\_Set\_Bits · 183  
P2\_SSI\_Set\_Clock · 184  
P2\_SSI\_Set\_Delay · 185  
P2\_SSI\_Start · 186  
P2\_SSI\_Status · 187  
P2\_Sync\_All · 16

## CNT-I Rev. E

- C:** P2\_Check\_LED · 5  
P2\_Cnt\_Clear · 162  
P2\_Cnt\_Enable · 163  
P2\_Cnt\_Get\_PW · 165  
P2\_Cnt\_Get\_PW\_HL · 166  
P2\_Cnt\_Get\_Status · 164  
P2\_Cnt\_Latch · 167  
P2\_Cnt\_Mode · 168  
P2\_Cnt\_PW\_Enable · 170  
P2\_Cnt\_PW\_Latch · 171  
P2\_Cnt\_Read · 172  
P2\_Cnt\_Read4 · 173  
P2\_Cnt\_Read\_Int\_Register · 174  
P2\_Cnt\_Read\_Latch · 175  
P2\_Cnt\_Read\_Latch4 · 176  
P2\_Cnt\_Sync\_Latch · 177
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- S:** P2\_Set\_LED · 6  
P2\_Sync\_All · 16

## CNT-T Rev. E

- C:** P2\_Check\_LED · 5  
P2\_Cnt\_Clear · 162  
P2\_Cnt\_Enable · 163  
P2\_Cnt\_Get\_PW · 165  
P2\_Cnt\_Get\_PW\_HL · 166  
P2\_Cnt\_Get\_Status · 164  
P2\_Cnt\_Latch · 167  
P2\_Cnt\_Mode · 168  
P2\_Cnt\_PW\_Enable · 170  
P2\_Cnt\_PW\_Latch · 171  
P2\_Cnt\_Read · 172  
P2\_Cnt\_Read4 · 173  
P2\_Cnt\_Read\_Int\_Register · 174  
P2\_Cnt\_Read\_Latch · 175  
P2\_Cnt\_Read\_Latch4 · 176  
P2\_Cnt\_Sync\_Latch · 177
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- S:** P2\_Set\_LED · 6  
P2\_Sync\_All · 16

## CPU-T11

- C:** CPU\_Digin · 12  
CPU\_Digout · 13  
CPU\_Dig\_IO\_Config · 14  
CPU\_Event\_Config · 15  
P2\_Check\_LED · 5
- S:** P2\_Set\_LED · 6

## DIO-32 Rev. E

- C:** P2\_Check\_LED · 5
- D:** P2\_Digin\_Edge · 136  
P2\_Digin\_FIFO\_Clear · 137  
P2\_Digin\_FIFO\_Enable · 138  
P2\_Digin\_FIFO\_Full · 140  
P2\_Digin\_FIFO\_Read · 141  
P2\_Digin\_Fifo\_Read\_Fast · 143  
P2\_Digin\_FIFO\_Read\_Timer · 145  
P2\_Digin\_Long · 146  
P2\_Digout · 147  
P2\_Digout\_Bits · 148  
P2\_Digout\_Long · 156  
P2\_Digout\_Reset · 157  
P2\_Digout\_Set · 158  
P2\_Digprog · 159  
P2\_Dig\_Latch · 133  
P2\_Dig\_Read\_Latch · 134  
P2\_Dig\_Write\_Latch · 135
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- G:** P2\_Get\_Digout\_Long · 160
- S:** P2\_Set\_LED · 6  
P2\_Sync\_All · 16

**DIO-32-TiCo Rev. E**

- C:** P2\_Check\_LED · 5
- D:** P2\_Digin\_Edge · 136  
P2\_Digin\_FIFO\_Clear · 137  
P2\_Digin\_FIFO\_Enable · 138  
P2\_Digin\_FIFO\_Full · 140  
P2\_Digin\_FIFO\_Read · 141  
P2\_Digin\_Fifo\_Read\_Fast · 143  
P2\_Digin\_FIFO\_Read\_Timer · 145  
P2\_Digin\_Long · 146  
P2\_Digout · 147  
P2\_Digout\_Bits · 148  
P2\_Digout\_FIFO\_Clear (Rev. E03) · 149  
P2\_Digout\_FIFO\_Empty (Rev. E03) · 150  
P2\_Digout\_FIFO\_Enable (Rev. E03) · 151  
P2\_Digout\_FIFO\_Read\_Timer (Rev. E03) · 152  
P2\_Digout\_FIFO\_Start (Rev. E03) · 153  
P2\_Digout\_FIFO\_Write (Rev. E03) · 154  
P2\_Digout\_Long · 156  
P2\_Digout\_Reset · 157  
P2\_Digout\_Set · 158  
P2\_Digprog · 159  
P2\_Dig\_FIFO\_Mode (Rev. E03) · 132  
P2\_Dig\_Latch · 133  
P2\_Dig\_Read\_Latch · 134  
P2\_Dig\_Write\_Latch · 135
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- G:** P2\_Get\_Digout\_Long · 160
- S:** P2\_Set\_LED · 6  
P2\_Sync\_All · 16

**EtherCAT-SL Rev. E**

- C:** P2\_Check\_LED · 5
- I:** P2\_ECAT\_Init · 270
- R:** P2\_ECAT\_Get\_State · 269  
P2\_ECAT\_Get\_Version · 268  
P2\_ECAT\_Read\_Data\_16L · 272  
P2\_ECAT\_Write\_Data\_16L · 273
- S:** P2\_Set\_LED · 6

**FlexRay-2 Rev. E**

- C:** P2\_Check\_LED · 5
- F:** P2\_FlexRay\_Get\_Version · 275  
P2\_FlexRay\_Init · 276  
P2\_FlexRay\_Read\_Word · 278  
P2\_FlexRay\_Reset · 279  
P2\_FlexRay\_Set\_LED · 280  
P2\_FlexRay\_Write\_Word · 281
- S:** P2\_Set\_LED · 6

**LIN-2 Rev. E**

- C:** P2\_Check\_LED · 5
- L:** P2\_LIN\_Get\_Version · 256  
P2\_LIN\_Init · 251  
P2\_LIN\_Init\_Apply · 254  
P2\_LIN\_Init\_Write · 253  
P2\_LIN\_Msg\_Transmit · 261  
P2\_LIN\_Msg\_Write · 259  
P2\_LIN\_Read\_Dat · 257  
P2\_LIN\_Reset · 255  
P2\_LIN\_Set\_LED · 262
- S:** P2\_Set\_LED · 6

**MIO-4 Rev. E**

- A:** P2\_ADC · 35  
P2\_ADC24 · 36  
P2\_ADC\_Read\_Limit · 37  
P2\_ADC\_Set\_Limit · 39
- C:** P2\_Check\_LED · 5
- D:** P2\_DAC · 117  
P2\_DAC4 · 118  
P2\_DAC4\_Packed · 119
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- M:** P2\_MIO\_Digin\_Long · 28  
P2\_MIO\_Digout · 29  
P2\_MIO\_Digout\_Long · 30  
P2\_MIO\_DigProg · 31  
P2\_MIO\_Dig\_Latch · 25  
P2\_MIO\_Dig\_Read\_Latch · 26  
P2\_MIO\_Dig\_Write\_Latch · 27  
P2\_MIO\_Get\_Digout\_Long · 32
- R:** P2\_Read\_ADC · 40  
P2\_Read\_ADC24 · 41  
P2\_Read\_ADC\_SConv · 42  
P2\_Read\_ADC\_SConv24 · 43
- S:** P2\_Seq\_Init · 45  
P2\_Seq\_Read · 48  
P2\_Seq\_Read24 · 49  
P2\_Seq\_Read\_Packed · 51  
P2\_Seq\_Start · 52  
P2\_Seq\_Wait · 53  
P2\_Set\_LED · 6  
P2\_Set\_Mux · 54  
P2\_SE\_Diff · 44  
P2\_Start\_Conv · 55  
P2\_Start\_DAC · 124  
P2\_Sync\_All · 16  
P2\_Sync\_Enable · 18  
P2\_Sync\_Stat · 22
- W:** P2\_Wait\_EOC · 56  
P2\_Wait\_Mux · 57  
P2\_Write\_DAC · 125  
P2\_Write\_DAC32 · 130  
P2\_Write\_DAC4 · 126  
P2\_Write\_DAC4\_Packed · 127

#### MIO-4-ET1 Rev. E

- A:** P2\_ADC · 35  
P2\_ADC24 · 36  
P2\_ADC\_Read\_Limit · 37  
P2\_ADC\_Set\_Limit · 39
- C:** P2\_Check\_LED · 5  
P2\_Cnt\_Clear · 162  
P2\_Cnt\_Enable · 163  
P2\_Cnt\_Get\_PW · 165  
P2\_Cnt\_Get\_PW\_HL · 166  
P2\_Cnt\_Get\_Status · 164  
P2\_Cnt\_Latch · 167  
P2\_Cnt\_Mode · 168  
P2\_Cnt\_PW\_Enable · 170  
P2\_Cnt\_PW\_Latch · 171  
P2\_Cnt\_Read · 172  
P2\_Cnt\_Read\_Int\_Register · 174  
P2\_Cnt\_Read\_Latch · 175  
P2\_Cnt\_Sync\_Latch · 177
- D:** P2\_DAC · 117  
P2\_DAC4 · 118  
P2\_DAC4\_Packed · 119
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- M:** P2\_MIO\_Digin\_Long · 28  
P2\_MIO\_Digout · 29  
P2\_MIO\_Digout\_Long · 30  
P2\_MIO\_DigProg · 31  
P2\_MIO\_Dig\_Latch · 25  
P2\_MIO\_Dig\_Read\_Latch · 26  
P2\_MIO\_Dig\_Write\_Latch · 27  
P2\_MIO\_Get\_Digout\_Long · 32
- R:** P2\_Read\_ADC · 40  
P2\_Read\_ADC24 · 41  
P2\_Read\_ADC\_SConv · 42  
P2\_Read\_ADC\_SConv24 · 43
- S:** P2\_Seq\_Init · 45  
P2\_Seq\_Read · 48  
P2\_Seq\_Read24 · 49  
P2\_Seq\_Read\_Packed · 51  
P2\_Seq\_Start · 52  
P2\_Seq\_Wait · 53  
P2\_Set\_LED · 6  
P2\_Set\_Mux · 54  
P2\_SE\_Diff · 44  
P2\_SSI\_Mode · 179  
P2\_SSI\_Read · 180  
P2\_SSI\_Set\_Bits · 183  
P2\_SSI\_Set\_Clock · 184  
P2\_SSI\_Set\_Delay · 185  
P2\_SSI\_Start · 186  
P2\_SSI\_Status · 187  
P2\_Start\_Conv · 55  
P2\_Start\_DAC · 124  
P2\_Sync\_All · 16  
P2\_Sync\_Enable · 18  
P2\_Sync\_Stat · 22

- W:** P2\_Wait\_EOC · 56  
P2\_Wait\_Mux · 57  
P2\_Write\_DAC · 125  
P2\_Write\_DAC32 · 130  
P2\_Write\_DAC4 · 126  
P2\_Write\_DAC4\_Packed · 127

#### OPT-16 Rev. E

- C:** P2\_Check\_LED · 5
- D:** P2\_Digin\_Edge · 136  
P2\_Digin\_FIFO\_Clear · 137  
P2\_Digin\_FIFO\_Enable · 138  
P2\_Digin\_FIFO\_Full · 140  
P2\_Digin\_FIFO\_Read · 141  
P2\_Digin\_Fifo\_Read\_Fast · 143  
P2\_Digin\_FIFO\_Read\_Timer · 145  
P2\_Digin\_Long · 146  
P2\_Dig\_Latch · 133  
P2\_Dig\_Read\_Latch · 134
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- S:** P2\_Set\_LED · 6  
P2\_Sync\_All · 16

#### OPT-32 Rev. E

- C:** P2\_Check\_LED · 5
- D:** P2\_Digin\_Edge · 136  
P2\_Digin\_FIFO\_Clear · 137  
P2\_Digin\_FIFO\_Enable · 138  
P2\_Digin\_FIFO\_Full · 140  
P2\_Digin\_FIFO\_Read · 141  
P2\_Digin\_Fifo\_Read\_Fast · 143  
P2\_Digin\_FIFO\_Read\_Timer · 145  
P2\_Digin\_Long · 146  
P2\_Dig\_Latch · 133  
P2\_Dig\_Read\_Latch · 134
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- S:** P2\_Set\_LED · 6  
P2\_Sync\_All · 16

#### Profi-SL Rev. E

- I:** P2\_Init\_Profibus · 264
- R:** P2\_Run\_Profibus · 266

#### Profi-SL Rev. E Rev. E

- C:** P2\_Check\_LED · 5
- S:** P2\_Set\_LED · 6

**PWM-16(-I) Rev. E**

- C:** P2\_Check\_LED · 5
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- P:** P2\_PWM\_Enable · 189  
P2\_PWM\_Get\_Status · 190  
P2\_PWM\_Init · 191  
P2\_PWM\_Latch · 193  
P2\_PWM\_Reset · 194  
P2\_PWM\_Standby\_Value · 195  
P2\_PWM\_Write\_Latch · 196  
P2\_PWM\_Write\_Latch\_Block · 197
- S:** P2\_Set\_LED · 6  
P2\_Sync\_All · 16  
P2\_Sync\_Enable · 18, · 22

**REL-16 Rev. E**

- C:** P2\_Check\_LED · 5
- D:** P2\_Digout · 147  
P2\_Digout\_Bits · 148  
P2\_Digout\_Long · 156  
P2\_Digout\_Reset · 157  
P2\_Digout\_Set · 158  
P2\_Dig\_Latch · 133  
P2\_Dig\_Write\_Latch · 135
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- G:** P2\_Get\_Digout\_Long · 160
- S:** P2\_Set\_LED · 6  
P2\_Sync\_All · 16

**RSxxx-2 Rev. E**

- C:** P2\_Check\_LED · 5  
P2\_Check\_Shift\_Reg · 240
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- G:** P2\_Get\_RS · 241
- R:** P2\_ · 247  
P2\_Read\_FIFO · 242  
P2\_RS485\_Send · 246  
P2\_RS\_Init · 243  
P2\_RS\_Reset · 245
- S:** P2\_Set\_LED · 6  
P2\_Set\_RS · 248
- W:** P2\_Write\_Fifo · 249

**RSxxx-4 Rev. E**

- C:** P2\_Check\_LED · 5  
P2\_Check\_Shift\_Reg · 240
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- G:** P2\_Get\_RS · 241
- R:** P2\_Read\_FIFO · 242  
P2\_RS485\_Send · 246  
P2\_RS\_Init · 243  
P2\_RS\_Reset · 245  
P2\_RS\_Set\_LED · 247
- S:** P2\_Set\_LED · 6  
P2\_Set\_RS · 248
- W:** P2\_Write\_Fifo · 249

**RTD-8 Rev. E**

- C:** P2\_Check\_LED · 5
- R:** P2\_RTD\_Channel\_Config · 200  
P2\_RTD\_Config · 202  
P2\_RTD\_Convert · 203  
P2\_RTD\_Read · 204  
P2\_RTD\_Read8 · 205  
P2\_RTD\_Start · 206  
P2\_RTD\_Status · 208
- S:** P2\_Set\_LED · 6

**TC-8-ISO Rev. A**

- T:** P2\_TC\_Read\_Latch · 210  
P2\_TC\_Read\_Latch4 · 212  
P2\_TC\_Read\_Latch8 · 214

**TC-8-ISO Rev. E**

- T:** P2\_TC\_Latch · 209  
P2\_TC\_Set\_Rate · 216

**TRA-16 Rev. E**

- C:** P2\_Check\_LED · 5
- D:** P2\_Digout · 147  
P2\_Digout\_Bits · 148  
P2\_Digout\_Long · 156  
P2\_Digout\_Reset · 157  
P2\_Digout\_Set · 158  
P2\_Dig\_Latch · 133  
P2\_Dig\_Write\_Latch · 135
- E:** P2\_Event\_Config · 8  
P2\_Event\_Enable · 7  
P2\_Event\_Read · 11
- G:** P2\_Get\_Digout\_Long · 160
- S:** P2\_Set\_LED · 6  
P2\_Sync\_All · 16

## A.3 Thematic Instruction List

The instructions are divided into the following groups. Inside a group the instructions are sorted alphabetically.

- Analog Inputs (fast ADC): page A-13
- Analog Inputs (multiplexer): page A-14
- Analog Outputs: page A-15
- CAN bus: page A-15
- Counters: page A-16
- Digital Inputs/Outputs: page A-16
- EtherCAT: page A-16
- FlexRay: page A-17
- LIN bus: page A-17
- Multi-I/O: page A-17
- Profibus: page A-17
- PWM outputs: page A-17
- RSxxx: page A-18
- SSI decoder: page A-18
- System: page A-18
- Temperature Inputs: page A-18

### Analog Inputs (fast ADC)

- P2\_ADCF executes a complete measurement on a Fast-ADC. The return value has a resolution of 16 bit.
- P2\_ADCF24 executes a complete measurement on a Fast-ADC. The return value has a resolution of 24 bit.
- P2\_ADCF\_Mode sets the working mode for all channels of the selected modules.
- P2\_ADCF\_Read\_Limit reads the limit-overflow and -underrun flags of all F-ADCs on the specified module.
- P2\_ADCF\_Read\_Min\_Max4 returns the minimum and maximum values of 4 F-ADC of the specified module in an array.
- P2\_ADCF\_Read\_Min\_Max8 gibt die Minimal- und Maximalwerte von 8 F-ADC des angegebenen Moduls in einem Feld zurück.
- P2\_ADCF\_Reset\_Min\_Max resets the minimum and maximum values of selected channels of the specified module.
- P2\_ADCF\_Set\_Limit sets the upper and lower limit for one F-ADC of the specified module.
- P2\_Burst\_CRead\_Unpacked1 copies an amount of the last measured values of a channel from the memory of the specified module into an array.
- P2\_Burst\_CRead\_Unpacked2 copies an amount of the last measurement values of 2 channels from the memory of the specified module into 2 arrays.
- P2\_Burst\_CRead\_Unpacked4 copies an amount of the last measurement values of 4 channels from the memory of the specified module into 4 arrays.
- P2\_Burst\_CRead\_Unpacked8 copies an amount of the last measurement values of 8 channels from the memory of the specified module into 8 arrays.
- P2\_Burst\_Init sets the parameters for a burst-measurement sequence on the specified module.
- P2\_Burst\_Read copies 32-bit values from the memory of the specified module into a specified array.
- P2\_Burst\_Read\_Index returns the address in the module memory, where the last measurement values have been stored.
- P2\_Burst\_Read\_Unpacked1 copies the measurement values of a channel into a specified array.
- P2\_Burst\_Read\_Unpacked2 copies the measurement values of 2 channels from the memory of the specified module into 2 arrays.
- P2\_Burst\_Read\_Unpacked4 copies the measurement values of 4 channels from the memory of the specified module into 4 arrays.

P2_Burst_Read_Unpacked8	copies the measurement values of 8 channels from the memory of the specified module into 8 arrays.
P2_Burst_Reset	resets the data pointer of burst sequences on all specified modules.
P2_Burst_Start	starts the burst measurement sequence on all specified modules at the same time.
P2_Burst_Status	determines the number of burst measurements which are still to execute on the specified module.
P2_Burst_Stop	stops a running burst-measurement sequence on all specified modules at the same time.
P2_Read_ADCF	reads out the conversion result from an F-ADC of the specified module. The return value has a resolution of 16 bit.
P2_Read_ADCF24	reads out the conversion result from an F-ADC of the specified module. The return value has a resolution of 24 bit.
P2_Read_ADCF32	reads the conversion result from 2 consecutive F-ADCs of the specified module and returns them in a single 32-bit value.
P2_Read_ADCF4	reads out the conversion results from the first 4 F-ADC of the specified module.
P2_Read_ADCF4_24B	reads out the conversion results from the first 4 F-ADC of the specified module. The return values have a resolution of 24 bits.
P2_Read_ADCF4_Packed	reads out the conversion results from the first 4 F-ADC of the specified module.
P2_Read_ADCF8	reads out the conversion results from all 8 F-ADCs of the specified module.
P2_Read_ADCF8_24B	reads out the conversion results from all 8 F-ADC of the specified module. The return values have a resolution of 24 bits.
P2_Read_ADCF8_Packed	reads out the conversion results from all 8 F-ADC of the specified module.
P2_Read_ADCF_SConv	reads the conversion result from an F-ADC of the specified module and immediately starts a new conversion.
P2_Read_ADCF_SConv24	reads the conversion result from an F-ADC of the specified module and immediately starts a new conversion.
P2_Read_ADCF_SConv32	reads the conversion results from 2 F-ADCs of the specified module and returns them in a 32-bit value.
P2_Set_Average_Filter	determines if the module calculates an average and of how many values the average is calculated.
P2_Set_Gain	sets the operating mode of a channel on the selected module and thus the gain and measurement range.
P2_Start_ConvF	starts the conversion on one or more F-ADCs of the specified module.
P2_Wait_EOFC	waits until the end of conversion on all F-ADCs of the specified module.

### Analog Inputs (multiplexer)

P2_ADC	runs a complete conversion on an ADC of the specified module. The return value has a resolution of 16 bit.
P2_ADC24	runs a complete conversion on an ADC of the specified module. The return value is formatted to 24 bit.
P2_ADC_Read_Limit	returns the flags of limit-overflow and -underrun from 16 ADCs of the specified module.
P2_ADC_Set_Limit	sets the upper and lower limit for one F-ADC of the specified module.
P2_Read_ADC	reads out the conversion result from an ADC of the specified module. The return value has a resolution of 16 bit.
P2_Read_ADC24	returns the conversion result from an ADC of the specified module. The return value has a resolution of 24 bit.
P2_Read_ADC_SConv	reads out the conversion result from an ADC of the specified module and immediately starts a new conversion.
P2_Read_ADC_SConv24	reads the conversion result from an ADC of the specified module and immediately starts a new conversion.
P2_Seq_Init	initializes the sequential control of the specified module.
P2_Seq_Read	reads a given number of values (16 Bit) from the specified module and copies them into a destination array.
P2_Seq_Read24	reads a given number of values (18 Bit) from the specified module and copies them into a destination array.
P2_Seq_Read_Packed	reads an even number of value pairs (16 Bit) from the specified module and copies them into a destination array.

P2_Seq_Start	starts the sequence control on all selected modules at the same time.
P2_Seq_Wait	waits until the sequence control has converted and stored all channels of the channel group on the specified module.
P2_Set_Mux	sets the multiplexer of the specified module to the selected input and to the selected gain.
P2_SE_Diff	sets the operating mode single ended or differential for all analog inputs of the specified module.
P2_Start_Conv	starts the conversion on the specified module.
P2_Wait_EOC	waits for the end of conversion on the specified module.
P2_Wait_Mux	waits for the end of the multiplexer settling on the specified module.

## Analog Outputs

P2_DAC	outputs an (analog) voltage on the channel of the specified module, that corresponds to the indicated digital value.
P2_DAC4	outputs 4 digital values from an array on the DAC 1...4 of the specified module as (analog) voltages.
P2_DAC4_Packed	outputs 4 packed digital values from an array on the DAC 1...4 of the specified module as (analog) voltages.
P2_DAC8	outputs 8 digital values from an array on the DAC 1...8 of the specified module as (analog) voltages.
P2_DAC8_Packed	outputs 8 packed digital values from an array on the DAC 1...8 of the specified module as (analog) voltages.
P2_Start_DAC	starts the conversion or output of all DAC on the specified module
P2_Write_DAC	writes a digital value into the output register of a DAC on the specified module.
P2_Write_DAC32	copies two 16 Bit values from a 32 Bit value into the output registers of a DAC pair of the specified module.
P2_Write_DAC4	writes 4 digital values from an array into the output registers of the DAC 1...4 of the specified module.
P2_Write_DAC4_Packed	writes 4 packed digital values from an array into the output registers of the DAC 1...4 of the specified module.
P2_Write_DAC8	writes 8 digital values from an array into the output registers of the DAC 1...8 of the specified module.
P2_Write_DAC8_Packed	writes 8 packed digital values from an array into the output registers of the DAC 1...8 of the specified module.

## CAN bus

CAN_Msg	is a one-dimensional array consisting of 9 elements, where the message objects of the CAN bus are saved during sending and receiving.
P2_CAN_Interrupt_Source	returns the CAN channels which have generated an event signal (interrupt).
P2_CAN_Set_LED	switches the additional LED of a CAN channel on (with color) or off.
P2_En_Interrupt	configures a message object of the specified module to generate an event signal (interrupt) when a message arrives.
P2_En_Receive	enables a message object on the specified module to receive messages.
P2_En_Transmit	enables a message object on the specified module to transmit messages.
P2_Get_CAN_Reg	returns the contents of a specified register on a CAN controller on the specified module.
P2_Init_CAN	initializes one of the CAN controllers on the specified module and sets it into an initial status.
P2_Read_Msg	returns the information if a new message in a message object of one of the CAN controllers on the module has been received.
P2_Read_Msg_Con	returns the information if a new message in a message object of one of the CAN controllers on the module has been received.
P2_Set_CAN_Baudrate	sets the baud rate on one of the controllers on the specified module and returns the status information.
P2_Set_CAN_Reg	writes a value in a register of the selected CAN controller on the specified module.
P2_Transmit	reads the data from the array CAN_Msg. As soon as the message object in one of the CAN controllers has access rights to the CAN bus, the message is sent.



## Counters

P2_Cnt_Clear	sets the counter values of one or more counters to 0 (zero), according to a given bit pattern.
P2_Cnt_Enable	enables or disables the counters selected by pattern.
P2_Cnt_Get_PW	returns frequency and duty cycle of a PWM counter.
P2_Cnt_Get_PW_HL	returns a stored high and low time of a PWM counter.
P2_Cnt_Get_Status	returns the counter status register of one counter.
P2_Cnt_Latch	transfers the current counter values of one or more counters into the relevant Latch A, depending on the bit pattern.
P2_Cnt_Mode	defines the operating mode of one counter.
P2_Cnt_PW_Enable	enables or disables the counters selected by pattern.
P2_Cnt_PW_Latch	copies the value of one or more PWM counters into a buffer.
P2_Cnt_Read4	transfers a current counter value into Latch A and returns the value.
P2_Cnt_Read	transfers a current counter value into Latch A and returns the value.
P2_Cnt_Read_Int_Register	returns the content of a counter register.
P2_Cnt_Read_Latch4	returns the value of a counter's Latch A.
P2_Cnt_Read_Latch	returns the value of a counter's Latch A.
P2_Cnt_Sync_Latch	copies the contents of selected counters and PWM counters into latches.

## Digital Inputs/Outputs

P2_Digin_Long	returns the status of the inputs (bits 31...00) of the specified module as bit pattern.
P2_Digout	sets a single output of the specified module to the level "high" or "low". All other outputs remain unchanged.
P2_Digout_Bits	sets the specified outputs of the specified module to the levels "high" or "low".
P2_Digout_Long	sets or clears all outputs on the specified module.
P2_Digout_Reset	sets the specified outputs of the specified module to the level "low".
P2_Digout_Set	sets the specified outputs of the specified module to the level "high".
P2_Digprog	programs the digital channels 0...31 of the specified module as inputs or outputs in groups of 8.
P2_Dig_Latch	transfers digital information from the inputs to the input latches and from the output latches to the outputs on the specified module.
P2_Dig_Read_Latch	returns the bits from the latch register for the digital inputs of the specified module.
P2_Dig_Write_Latch	writes a 32 bit value into the latch register for the digital outputs on the specified module.
P2_Get_Digout_Long	returns the contents of the output latch (register for digital outputs) on the specified module.
P2_Digin_Edge	returns whether a positive or negative edge has occurred on digital inputs of the specified module.
P2_Digin_FIFO_Clear	clears the FIFO of the edge detection unit on the specified module.
P2_Digin_FIFO_Enable	determines, which input channels of the specified module the edge detection unit will monitor.
P2_Digin_FIFO_Full	returns the number of saved value pairs in the FIFO of the edge detection unit.
P2_Digin_FIFO_Read	reads the value pairs from the FIFO of the edge detection unit and writes them into 2 arrays.
P2_Digin_Fifo_Read_Fast	P2_Digin_FIFO_Read reads the value pairs from the FIFO of the edge detection unit and writes them into 2 arrays.
P2_Digin_FIFO_Read_Timer	returns the current status of the 100MHz timer on the specified module.
P2_Digout_FIFO_Clear	stops the edge output and clears the edge output FIFO on the specified module.
P2_Digout_FIFO_Empty	returns the number of free value pairs in the edge output FIFO.
P2_Digout_FIFO_Enable	sets the output channels of the specified module where edges are output.
P2_Digout_FIFO_Read_Timer	returns the current value of the 100MHz counter on the specified module.
P2_Digout_FIFO_Start	starts the edge output on the specified module.
P2_Digout_FIFO_Write	writes value pairs into the output edge FIFO.
P2_Dig_FIFO_Mode	sets the FIFO operation mode on the specified module, input with edge detection or edge output.

## EtherCAT

P2_ECAC_Get_State	P2_ECAC_Get_Version	returns the version of the EtherCAT interface.
-------------------	---------------------	--



P2\_ECAT\_Get\_Version returns the version of the EtherCAT interface.  
P2\_ECAT\_Init initializes the EtherCAT Slave.  
P2\_ECAT\_Read\_Data\_16L P2\_ECAT\_Get\_Version returns the version of the EtherCAT interface.  
P2\_ECAT\_Write\_Data\_16L P2\_ECAT\_Get\_Version returns the version of the EtherCAT interface.

## FlexRay

P2\_FlexRay\_Get\_Version returns the version number of the FlexRay interface.  
P2\_FlexRay\_Init initializes the data transfer between ADwin CPU and the FlexRay interface on a specified module.  
P2\_FlexRay\_Read\_Word returns a 16 bit value from a FlexRay controller on the specified module.  
P2\_FlexRay\_Reset resets a FlexRay controller on the specified module.  
P2\_FlexRay\_Set\_LED switches a channel LED of a FlexRay controller on the specified module on or off.  
P2\_FlexRay\_Write\_Word writes a 16 bit value to an address in a FlexRay controller of the specified module.

## LIN bus

P2\_LIN\_Get\_Version returns the version number of the LIN interface.  
P2\_LIN\_Init initializes the data transfer between ADwin CPU and the LIN interface on a specified module.  
P2\_LIN\_Init\_Apply activates the initialization data given with P2\_LIN\_Init\_Write for all LIN interfaces.  
P2\_LIN\_Init\_Write sets baudrate and operating mode for a specified LIN interface.  
P2\_LIN\_Msg\_Transmit sends a header to the LIN bus. To use only with operating mode LIN master.  
P2\_LIN\_Msg\_Write configures a message box of a LIN interface for send or receive.  
P2\_LIN\_Read\_Dat reads the data of a message box or the status of a LIN interface and writes the result into an array.  
P2\_LIN\_Reset resets all LIN interfaces, either all settings (start-up status) or LIN-internal counters only.  
P2\_LIN\_Set\_LED switches the additional LED of a LIN interface on (with color) or off.

## Multi-I/O

P2\_MIO\_Digin\_Long returns the status of the inputs (bits 7...0) of the specified module as bit pattern  
P2\_MIO\_Digout sets a single output of the specified module to the level "high" or "low". All other outputs remain unchanged.  
P2\_MIO\_Digout\_Long sets or clears all outputs on the specified module.  
P2\_MIO\_DigProg programs the digital channels 0...7 of the specified module as inputs or outputs in groups of 4.  
P2\_MIO\_Dig\_Latch transfers digital information from the inputs to the input latches and/or from the output latches to the outputs on the specified module.  
P2\_MIO\_Dig\_Read\_Latch returns the bits from the latch register for the digital inputs of the specified module.  
P2\_MIO\_Dig\_Write\_Latch writes a 32 bit value into the latch register for the digital outputs on the specified module.  
P2\_MIO\_Get\_Digout\_Long returns the contents of the output latch (register for digital outputs) on the specified module.

## Profibus

P2\_Init\_Profibus initializes the Profibus Slave.  
P2\_Run\_Profibus exchanges data with the Profibus Slave.

## PWM outputs

P2\_PWM\_Enable enables or disables one or more PWM outputs.  
P2\_PWM\_Get\_Status returns the operation status of all PWM outputs.  
P2\_PWM\_Init sets the defaults for one PWM output.  
P2\_PWM\_Latch enables frequency and duty cycle of one or more PWM outputs to be output.  
P2\_PWM\_Reset stops the output of one or more PWM outputs immediately..  
P2\_PWM\_Standby\_Value sets the default TTL levels for all PWM outputs.  
P2\_PWM\_Write\_Latch writes frequency and duty cycle into the latch register.  
P2\_PWM\_Write\_Latch\_Block writes frequency and duty cycle for several PWM outputs into the latch registers.

**RSxxx**

P2_Check_Shift_Reg	returns, if all data has been sent, which was written into the send-FIFO of the channel on the specified module.
P2_Get_RS	reads out the controller register on the specified module.
P2_Read_FIFO	reads a value from the input FIFO of a specified channel on the specified module.
P2_RS485_Send	determines the transfer direction for a specified channel on the specified module.
P2_RS_Init	initializes one channel on the specified module.
P2_RS_Reset	executes a hardware reset on the specified module and deletes the settings for all channels.
P2_RS_Set_LED	switches the additional LED of a RSxxx channel on (with color) or off.
P2_Set_RS	writes a value into a specified register on the specified module.
P2_Write_Fifo	P2_Write_FIFO writes a value into the send-FIFO of a specified channel on the specified module.

**SSI decoder**

P2_SSI_Mode	sets the modes of all SSI decoders on the specified module, either "single shot" (read out once) or "continuous" (read out continuously).
P2_SSI_Read	returns the last saved counter value of a specified SSI counter on the specified module.
P2_SSI_Read2	returns the last saved counter values of both SSI counters on the specified module.
P2_SSI_Set_Bits	sets for an SSI counter on the specified module the amount of bits which generate a complete encoder value.
P2_SSI_Set_Clock	sets the clock rate (approx. 100kHz to 2.5MHz) on the specified module, with which the decoder is clocked.
P2_SSI_Set_Delay	sets the waiting time between reading two encoder values for one SSI-decoder on the specified module.
P2_SSI_Start	starts the reading of one or both SSI decoders on the specified module (only in mode "single shot").
P2_SSI_Status	returns the current read-status on the specified module for a specified decoder.

**System**

P2_Check_LED	returns the status of the LED (on top of the front panel) of the module.
CPU_Digin (T11)	Processor T11 only. CPU_Digin returns, whether a falling edge arose at the input DIG I/O of the processor module since the last call of the instruction.
CPU_Digout	sets a DIG I/O output of the processor module to the selected TTL level.
CPU_Dig_IO_Config	configures all DIG I/O channels of the processor module.
CPU_Event_Config	configures the Event In channel of the processor module.
P2_Event2_Config	configures the pre-processing of event signals on the specified module.
P2_Event_Config	configures the external event input of the specified module.
P2_Event_Enable	enables or disables an external event input on the specified module.
P2_Event_Read	returns the current TTL level at the event inputs of the specified module.
P2_Sync_Mode	enables or disables the synchronization (of conversions) with other modules as master or as slave.
P2_Set_LED	switches the LED (on top of the front panel) on or off.
P2_Sync_All	starts a specified action synchronically on the selected modules.
P2_Sync_Enable	enables or disables the synchronizing option for selected inputs, outputs or function groups on the specified module.
P2_Sync_Stat	returns the settings of the synchronizing option of the specified module.

**Temperature Inputs**

P2_RTD_Channel_Config	sets the temperature measuring mode for a certain channel on the specified module.
P2_RTD_Config	initializes the temperature measurement on the specified module.
P2_RTD_Convert	calculates the resistance or the temperature in degrees Celsius or Fahrenheit from the measured digital value of a temperature sensor.

P2_RTD_Read	returns the current digital measurement value of a temperature sensor at the specified channel on the module.
P2_RTD_Read8	returns the current digital measurement values of temperature sensors at all channels on the module.
P2_RTD_Start	starts the temperature measurement cycle on all specified modules at the same time.
P2_RTD_Status	returns the status of temperature measurement cycle in "single shot" mode on the specified module.
P2_TC_Latch	copies the current voltage values at the inputs into latches.
P2_TC_Read_Latch	returns the thermo voltage ( $\mu\text{V}$ ) or temperature ( $^{\circ}\text{C}$ / $^{\circ}\text{F}$ ) of the selected channel of the module.
P2_TC_Read_Latch4	returns the thermo voltage ( $\mu\text{V}$ ) or temperature ( $^{\circ}\text{C}$ / $^{\circ}\text{F}$ ) of the channels 1...4 of the module.
P2_TC_Read_Latch8	returns the thermo voltage ( $\mu\text{V}$ ) or temperature ( $^{\circ}\text{C}$ / $^{\circ}\text{F}$ ) of the channels 1...8 of the module.
P2_TC_Set_Rate	sets the sampling rate of the selected module.