

# ***ADwin-Pro, Pro II***

**System specifications**

**Programming in *ADbasic***

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

Hotline: +49 6251 96320  
Fax: +49 6251 56819  
E-Mail: [info@ADwin.de](mailto:info@ADwin.de)  
Internet: [www.ADwin.de](http://www.ADwin.de)



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

## Table of contents

Typographical Conventions .....	IV
1 Introduction .....	1
2 The Program ADpro.exe .....	2
3 ADbasic instruction for ADwin-Pro I modules .....	3
3.1 Pro I: All Modules .....	3
3.2 Pro I: Input Modules .....	17
3.3 Pro I: Output Modules .....	72
3.4 Pro I: Digital Modules .....	86
3.5 Pro I: Special Modules .....	163
4 ADbasic instruction for ADwin-Pro II modules .....	221
4.1 Pro II: All Modules .....	221
4.2 Pro II: Input Modules .....	238
4.3 Pro II: Output Modules .....	311
5 Program Examples .....	326
5.1 Online Evaluation of Measurement Data .....	326
5.2 Digital Proportional Controller .....	326
5.3 Data Exchange with DATA arrays .....	327
5.4 Digital PID Controller .....	327
5.5 Data exchange with fieldbus .....	330
5.6 Examples for RS232 and RS485 .....	331
5.7 Continuous signal conversion .....	335
Annex .....	A-1
A.1 Alphabetic Instruction List .....	A-1
A.2 Instruction List sorted by Module Types .....	A-5
A.3 Thematic Instruction List .....	A-17

## 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 have absolutely to be considered in order to guarantee an operation without any errors
- 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* 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. ([Instruction List sorted by Module Types](#) see annex).

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* 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* 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.

Subject to change.

Hotline address: see inner side of cover page.



#### Qualified personnel

#### Availability of the documents



#### Legal information

## 2 The Program ADpro.exe

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

- Show the modules on an ADwin-Pro system as well as information on the modules.

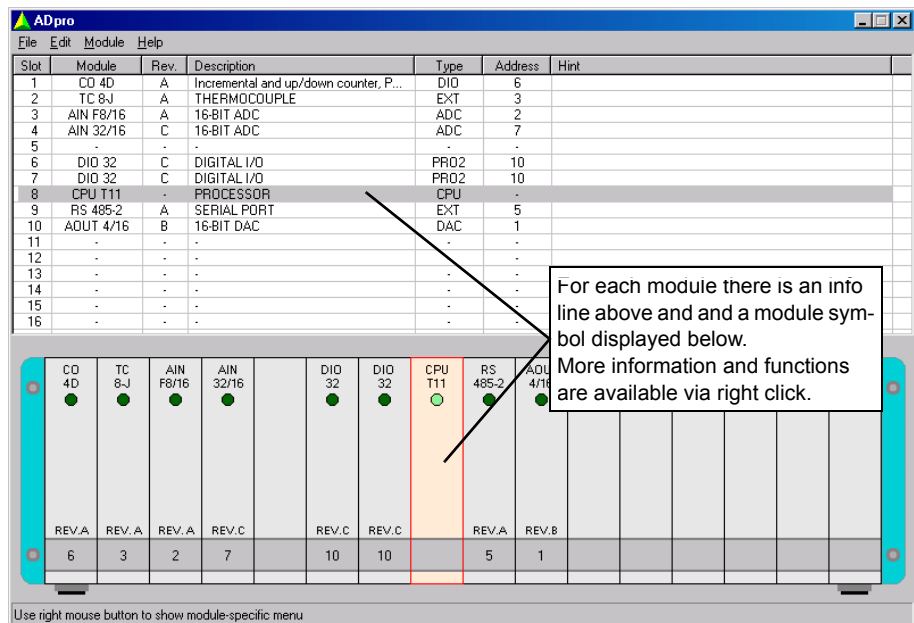
- Set the module address for Pro II modules (see hardware manual).

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 <.NET Framework 1.1> is installed on the PC.

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

The instructions for the access to the *ADwin-Pro* modules are to be found in the include files. The instructions are sorted according to the include files 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](#)

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.

#### 3.1 Pro I: All Modules

The include file `<ADWINPRO.INC>` includes general system functions and procedures that are necessary to get access to the *ADwin-Pro I* modules. If you include this file with the *ADbasic* instruction

```
#INCLUDE ADwinPRO_ALL.inc
```

in your *ADbasic* process, you will be able to apply all functions and procedures of this file.

Some instructions for Pro I modules need the parameter `mod_class` that determines the module class. Use the following constants for this parameter, which are defined in the file `<ADWINPRO.INC>`:

dio	= 000h	For all digital input/output or counter modules
ad	= 040h	For all analog/digital converter modules
da	= 080h	For all digital/analog converter modules
cpu	= 0A0h	For the processor module
ext	= 0C0h	For all other modules



## CHECKLED

**CHECKLED** returns the status of the green LED (on top of the front panel) of the module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = CHECKLED(module,mod_class)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>mod_class</code>	Module class: One of the constants <code>ad</code> , <code>da</code> , <code>dio</code> , <code>cpu</code> or <code>ext</code> .	LONG
<code>ret_val</code>	0: LED off (default). 1: LED on.	LONG

### Notes

On some modules there are additional LEDs whereof the status cannot be returned with this instruction.

### See also

[SETLED](#)

### To be used for the modules

(LP)SH-8(-FI), Aln-16/14-C Rev. A, Aln-32/12 Rev. A, Aln-32/12 Rev. B, Aln-32/14 Rev. A, Aln-32/16 Rev. B, Aln-32/16 Rev. C, Aln-8/12 Rev. A, Aln-8/12 Rev. B, Aln-8/14 Rev. A, Aln-8/16 Rev. A, Aln-8/16 Rev. B, Aln-8/16 Rev. C, Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A, AOut-16/8-12, AOut-4/16 Rev. A, AOut-4/16 Rev. B, AOut-4/16 Rev. C, AOut-8/16 Rev. A, AOut-8/16 Rev. B, AOut-8/16 Rev. C, CAN-1, CAN-2, CNT-16/16(-I), CNT-16/32(-I), CNT-8/32(-I), CNT-PW4(-I), CNT-VR4(-I), CNT-VR4L(-I), CO4, CPU-T10, CPU-T9, DIO-32, DIO-32 Rev. B, INTER-SL, LS2 Rev. A, OPT-16 Rev. A, OPT-16 Rev. B, Profi-SL, PT100-4, PT100-8, PWM-4(-I), REL-16 Rev. A, REL-16 Rev. B, RS232-2, RS232-4, RS485-2, RS485-4, Storage Rev. A, TC-16, TC-4, TC-8, TRA-16 Rev. A, TRA-16 Rev. B

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
  IF (CHECKLED(1,dio)=0) THEN 'If LED is off ...
    SETLED(1,dio,1)           '... switch LED on
  ENDIF
```



Processor T9 and T10 only. **CPU\_DIGIN** returns, whether a falling edge arose at the input `Digin 0` of the processor module since the last call of the instruction.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = CPU_DIGIN()
```

## Parameters

<code>ret_val</code>	Flag, if a falling edge has been detected at the <code>Digin 0</code> : 0: No falling edge detected. 1: Falling edge has been detected at least once.
----------------------	---

## Notes

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

The input `Digin 0` works with TTL signals only.

## See also

[CPU\\_DIGIN \(T11\)](#)

## To be used for the modules

CPU-T10, CPU-T9

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM dummy AS LONG

INIT:
'Read and thus reset the flag
dummy = CPU_DIGIN()

EVENT:
...
IF(CPU_DIGIN() = 1) THEN 'If falling edge has been detected ...
END                      '... end this program
ENDIF
...
```

## CPU\_DIGIN

## EVENTENABLE

**EVENTENABLE** enables or disables an external event input on the module. With a signal at this input a cycle of an *ADbasic* process can be controlled.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

EVENTENABLE (module, mod_class, value)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>mod_class</code>	Module class: One of the constants <code>ad</code> , <code>da</code> , <code>dio</code> , <code>cpu</code> or <code>ext</code> .	LONG
<code>value</code>	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. As soon as you have enabled the event input with **EVENTENABLE**, the input signal will be forwarded to the processor module. The processor module recognizes a rising edge as event signal and the specified process responds.

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.

### To be used for the modules

CNT-16/16(-I), CNT-16/32(-I), CNT-8/32(-I), CNT-PW4(-I), CNT-VR4(-I), CNT-VR4L(-I), CO4, DIO-32, DIO-32 Rev. B, OPT-16 Rev. A, OPT-16 Rev. B, PWM-4(-I), REL-16 Rev. A, REL-16 Rev. B, TRA-16 Rev. A, TRA-16 Rev. B

### Example

```
#INCLUDE ADwinPRO_ALL.inc
INIT:
    'Enables an external event at the digital module 1
    EVENTENABLE (1,dio,1)

EVENT:
    ...
```



**RESETWATCHDOGTIMER** sets the watchdog counter of the CPU module to the start value. The counter remains enabled.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

RESETWATCHDOGTIMER()
```

## Notes

As soon as the watchdog counter is enabled, it decrements the counter values continuously. When the counter value reaches 0 (zero) the system assumes a malfunction and all modules are reset to their initial function (power-on status). Thus for instance, all programs are stopped, inputs and outputs as well as set points are reset.

Processor type	Duration
T9, T10, T11	1600ms

Counting Time from start value to 0

Set the active watchdog timer at least once to the start value within the counting interval, in order to keep your Pro system working. When the watchdog timer is disabled this instruction has no function.

The watchdog function is used as a monitoring device for the *ADwin-Pro* system.

## RESETWATCH DOGTIMER



## See also

[STARTWATCH DOG](#), [STOPWATCHDOG](#)

## To be used for the modules

CPU-T10, CPU-T11, CPU-T9

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
    STARTWATCHDOG ()           'Activate watchdog
EVENT:
    RESETWATCHDOGTIMER ()      'Reset watchdog continuously
...
FINISH:
    STOPWATCHDOG ()           'Disable watchdog
```

## SETLED

**SETLED** switches the green LED (on top of the front panel) on or off.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
SETLED (module, mod_class, value)
```

### Parameters

module	Specified module address (1...255).	LONG
mod_class	Module class: One of the constants ad, da, dio, cpu or ext.	LONG
value	Status of the LED: 0: switch off. 1: switch on.	LONG

### Notes

The Pro-Storage module has 3 LEDs with 2 colours on the front panel, whereof 2 are programmable with the instruction **SETLED**. The bits apply to the LEDs as follows:

Bits in <code>ret_val</code>	31:04	03:02	01:00
LED position	–	down right	top

For each LED status of the Pro-Storage module you have to set 2 bits:

Bit pattern	LED status
00b	LED off
01b	LED green
10b	LED red
11b	LED off

### See also

[CHECKLED](#)

### To be used for the modules

(LP)SH-8(-I), Aln-16/14-C Rev. A, Aln-32/12 Rev. A, Aln-32/12 Rev. B, Aln-32/14 Rev. A, Aln-32/16 Rev. B, Aln-32/16 Rev. C, Aln-8/12 Rev. A, Aln-8/12 Rev. B, Aln-8/14 Rev. A, Aln-8/16 Rev. A, Aln-8/16 Rev. B, Aln-8/16 Rev. C, Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A, AOut-16/8-12, AOut-4/16 Rev. A, AOut-4/16 Rev. B, AOut-4/16 Rev. C, AOut-8/16 Rev. A, AOut-8/16 Rev. B, AOut-8/16 Rev. C, CAN-1, CAN-2, CNT-16/16(-I), CNT-16/32(-I), CNT-8/32(-I), CNT-PW4(-I), CNT-VR4(-I), CNT-VR4L(-I), CO4, CPU-T10, CPU-T9, DIO-32, DIO-32 Rev. B, INTER-SL, LS2 Rev. A, OPT-16 Rev. A, OPT-16 Rev. B, Profi-SL, PT100-4, PT100-8, PWM-4(-I), REL-16 Rev. A, REL-16 Rev. B, RS232-2, RS232-4, RS485-2, RS485-4, Storage Rev. A, TC-16, TC-4, TC-8, TRA-16 Rev. A, TRA-16 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc
INIT:
    SETLED(1,cpu,1)      'Switch on LED of processor module 1
    SETLED(1,ad,1)       'Switch on LED of the A/D module 1
    SETLED(1,da,1)       'Switch on LED of the D/A module 1
    SETLED(1,dio,1)      'Switch on LED of the digital module 1
    SETLED(2,dio,0110b)  'Switch upper LED to red, lower LED to
                        'green on module Pro-Storage
                        '(DIO no.2)

EVENT:
    ...

FINISH:
    SETLED(1,cpu,0)      'Switch off LED of processor module 1
    SETLED(1,ad,0)       'Switch off LED of A/D module 1
    SETLED(1,da,0)       'Switch off LED of D/A module 1
    SETLED(1,dio,0)      'Switch off LED of digital module 1
    SETLED(2,dio,0)      'Switch off all LEDs of Pro-Storage
```

## STARTWATCH DOG

**STARTWATCHDOG** activates the watchdog counter of the CPU module and sets its start value.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
STARTWATCHDOG ( )
```

### Notes

As soon as the watchdog counter is enabled, it decrements the counter values continuously. When the counter value reaches 0 (zero) the system assumes a malfunction and all modules are reset to their initial function (power-on status). Thus for instance, all programs are stopped, inputs and outputs as well as set points are reset.

Processor type	Duration
T9, T10, T11	1600ms

Counting Time from start value to 0

Set the active watchdog timer at least once to the start value within the counting time, in order to keep your Pro system working (see **RESET-WATCHDOGTIMER**).

The watchdog function is used for monitoring the *ADwin-Pro* system.

### See also

[RESETWATCH DOGTIMER](#), [STOPWATCHDOG](#)

### To be used for the modules

CPU-T10, CPU-T11, CPU-T9

### Example

```
#INCLUDE ADwinPRO_ALL.inc  
INIT:  
  STARTWATCHDOG ( )           'Activate watchdog
```



**STOPWATCHDOG** disables the watchdog counter of the CPU module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
STOPWATCHDOG ()
```

## Notes

The watchdog function is used for monitoring the *ADwin-Pro* system.  
You can enable it again with **STARTWATCHDOG**.

## See also

[RESETWATCH DOGTIMER](#), [STARTWATCH DOG](#)

## To be used for the modules

CPU-T10, CPU-T11, CPU-T9

## Example

```
#INCLUDE ADwinPRO_ALL.inc  
INIT:  
    STARTWATCHDOG ()          'Activate watchdog  
    ...  
  
EVENT:  
    RESETWATCHDOGTIMER ()    'Reset watchdog  
    ...  
  
FINISH:  
    STOPWATCHDOG ()          'Disable watchdog
```

## STOPWATCHDOG



## SYNCALL

**SYNCALL** starts a specified action synchronically on all modules which have been activated before with **SYNCENABLE**.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
  
SYNCALL ()
```

### Notes

Depending on the module type, one of the following actions is (synchronously) started on all modules which have been activated by **SYNCENABLE**. The configurations you have made before for the multiplexer, output value or burst mode apply.

Module type	Action
Analog input	Start A/D conversion on all ADCs (for <b>SYNCENABLE</b> =1) or Start burst measurement sequence (for <b>SYNCENABLE</b> =3 and only for the modules Pro-AIn-F-x/14)
Analog output	Start D/A conversion with the value from the DAC register (see <b>WRITEDAC</b> )
Digital input	Transfer current status of the inputs into the input temporary register (read out the values there with <b>DIG_READLATCH1, D</b> ).
Digital output	Read the value from the output temporary register and transfer it to the digital output (see <b>DIG_WRITELATCH1, D</b> ).
Counter	Transfer current counter values into the counter temporary register (read out the values there with <b>CNT_ READLATCH16, C, CO4_READLATCH</b> ).

### See also

[SYNCENABLE](#), [SYNCSTAT](#)

### To be used for the modules

(LP)SH-8(-FI), AIn-16/14-C Rev. A, AIn-32/12 Rev. A, AIn-32/12 Rev. B, AIn-32/14 Rev. A, AIn-32/16 Rev. B, AIn-32/16 Rev. C, AIn-8/12 Rev. A, AIn-8/12 Rev. B, AIn-8/14 Rev. A, AIn-8/16 Rev. A, AIn-8/16 Rev. B, AIn-8/16 Rev. C, AIn-F-4/12 Rev. A, AIn-F-4/14 Rev. B, AIn-F-4/16 Rev. A, AIn-F-8/12 Rev. A, AIn-F-8/14 Rev. B, AIn-F-8/16 Rev. A, AOut-16/8-12, AOut-4/16 Rev. A, AOut-4/16 Rev. B, AOut-4/16 Rev. C, AOut-8/16 Rev. A, AOut-8/16 Rev. B, AOut-8/16 Rev. C, CNT-16/16(-I), CNT-16/32(-I), CNT-8/32(-I), CNT-PW4(-I), CNT-VR4(-I), CNT-VR4L(-I), CO4, DIO-32, DIO-32 Rev. B, OPT-16 Rev. A, OPT-16 Rev. B, PWM-4(-I), REL-16 Rev. A, REL-16 Rev. B, TRA-16 Rev. A, TRA-16 Rev. B



## Example

```
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADwinPRO_ALL.inc
DIM i AS LONG
DIM DATA_1[1000], DATA_2[1000], DATA_3[1000] AS LONG
INIT:
    REM Set the multiplexer of the A/D modules 1-3 to input 1
    SET_MUX(1,0)
    SET_MUX(2,0)
    SET_MUX(3,0)
    REM Enable synchronization of the A/D modules 1-3
    SYNCENABLE(1,ad,1)
    SYNCENABLE(2,ad,1)
    SYNCENABLE(3,ad,1)
    i=1                                'Initialize index

EVENT:
    REM Start conversion of all active modules synchronously
    SYNCALL()
    WAIT_EOC(1)                        'Wait for end of conversion
    DATA_1[i]=READADC(1)              'Read out A/D converter module 1
    DATA_2[i]=READADC(2)              'Read out A/D converter module 2
    DATA_3[i]=READADC(3)              'Read out A/D converter module 3
    IF (i=1000) THEN END               'End process after 1000 repetitions
    INC(i)                             'Increment index
```

## SYNCENABLE

**SYNCENABLE** enables or disables the synchronizing option on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
SYNCENABLE (module, mod_class, value)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>mod_class</code>	Module class: One of the constants <code>ad</code> , <code>da</code> , <code>dio</code> , <code>cpu</code> or <code>ext</code> .	LONG
<code>value</code>	Setting of the synchronizing option: 0 : disabled. 1: enabled (for "normal" action; see <a href="#">SYN-CALL</a> ). 3: enabled for burst measurement sequence (only for the modules Pro-Aln-F-x/14).	LONG

### Notes

The synchronizing option has to be enabled separately for each module. As many modules as you like can be synchronized.

The synchronizing signal is triggered by **SYNCALL**.

### See also

[SYNCALL](#), [SYNCSTAT](#)

### To be used for the modules

(LP)SH-8(-FI), Aln-16/14-C Rev. A, Aln-32/12 Rev. A, Aln-32/12 Rev. B, Aln-32/14 Rev. A, Aln-32/16 Rev. B, Aln-32/16 Rev. C, Aln-8/12 Rev. A, Aln-8/12 Rev. B, Aln-8/14 Rev. A, Aln-8/16 Rev. A, Aln-8/16 Rev. B, Aln-8/16 Rev. C, Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A, AOut-16/8-12, AOut-4/16 Rev. A, AOut-4/16 Rev. B, AOut-4/16 Rev. C, AOut-8/16 Rev. A, AOut-8/16 Rev. B, AOut-8/16 Rev. C, CNT-16/16(-I), CNT-16/32(-I), CNT-8/32(-I), CNT-PW4(-I), CNT-VR4(-I), CNT-VR4L(-I), CO4, DIO-32, DIO-32 Rev. B, OPT-16 Rev. A, OPT-16 Rev. B, PWM-4(-I), REL-16 Rev. A, REL-16 Rev. B, TRA-16 Rev. A, TRA-16 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADwinPRO_ALL.inc
#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:
  REM Set multiplexer of the A/D modules 1-3 to input 1
  SET_MUX(1,0)          'Set multiplexer to input 1
  SET_MUX(2,0)
  SET_MUX(3,0)
  REM Enable synchronization: A/D module 1, DIO modules 1+2
  SYNCENABLE(1,ad,1)
  SYNCENABLE(1,dio,1)
  SYNCENABLE(2,ad,1)
  i=1                    'Initialize index

EVENT:
  REM - Start conversion of A/D module 1
  REM - Transfer the current status of the dig. inputs of the
  REM   digital I/O module 1 into the input temporary register
  REM   or output the value of the output temporary register
  REM   to the digital outputs
  REM - Transfer the current counter values of the counters of
  REM   the modules 1 and 2 into the counter temporary register
  SYNCALL()              'Start conversion of the A/D module
  WAIT_EOC(1)            'Wait for end of conversion
  DATA_1[i]=READADC(1)   'Read out A/D converter module 1
  REM Read out temporary register of DIO modules
  DATA_2[i]=DIG_READLATCH1(1)
  DATA_3[i]=CNT_READ32(2,1) 'Temporary register of counter 1
  DATA_4[i]=CNT_READ32(2,2) 'Temporary register of counter 2
  IF (i=1000) THEN END    'End process after 1000 repetitions
  INC(i)                  'Increment index
```

## SYNCSTAT

**SYNCSTAT** returns the settings of the synchronizing option of the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = SYNCSTAT(module,mod_class)
```

### Parameters

module	Specified module address (1...255).	LONG
mod_class	Module class: One of the constants ad, da, dio, cpu or ext.	LONG
ret_val	Setting of the synchronizing option: 0 : disabled. 1 : enabled (for "normal" actions; see <a href="#">SYNCALL</a> ). 3 : enabled for burst-measurement sequences (only for the modules Pro-Aln-F-x/14).	LONG

### See also

[SYNCALL](#), [SYNCENABLE](#)

### To be used for the modules

(LP)SH-8(-FI), Aln-16/14-C Rev. A, Aln-32/12 Rev. A, Aln-32/12 Rev. B, Aln-32/14 Rev. A, Aln-32/16 Rev. B, Aln-32/16 Rev. C, Aln-8/12 Rev. A, Aln-8/12 Rev. B, Aln-8/14 Rev. A, Aln-8/16 Rev. A, Aln-8/16 Rev. B, Aln-8/16 Rev. C, Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A, AOut-16/8-12, AOut-4/16 Rev. A, AOut-4/16 Rev. B, AOut-4/16 Rev. C, AOut-8/16 Rev. A, AOut-8/16 Rev. B, AOut-8/16 Rev. C, CNT-16/16(-I), CNT-16/32(-I), CNT-8/32(-I), CNT-PW4(-I), CNT-VR4(-I), CNT-VR4L(-I), CO4, DIO-32, DIO-32 Rev. B, OPT-16 Rev. A, OPT-16 Rev. B, PWM-4(-I), REL-16 Rev. A, REL-16 Rev. B, TRA-16 Rev. A, TRA-16 Rev. B

### Example

```
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADwinPRO_ALL.inc
DIM i AS LONG
DIM DATA_1[1000], DATA_2[1000] AS LONG

INIT:
  IF (SYNCSTAT(1,da)=0) THEN 'If synchronizing option is disabled
    SYNCENABLE(1,da,1)      'then enable synchronization for
    SYNCENABLE(2,da,1)      'the D/A modules 1+2
  ENDIF
  i=1                        'Initialize index

EVENT:
  WRITEDAC(1,1,DATA_1[i]) 'Prepare output
  WRITEDAC(2,1,DATA_2[i])
  'Start output on both modules synchronously
  SYNCALL()
  IF (i=1000) THEN END      'End process after 1000 repetitions
  INC(i)                    'Increment index
```

### 3.2 Pro I: Input Modules

The include file `ADWPAD.INC` includes all functions and procedures that are necessary to get access to the *ADwin-Pro* I A/D-modules. If you include this file with the *ADbasic* command

```
#INCLUDE ADwinPRO_ALL.inc
```

in your *ADbasic* process you will be able to apply all functions and procedures of this file.

On the following pages all `ADWPAD.INC` functions will be described more detailed. In addition an easy example is presented for every function.

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

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



## ADC

**ADC** executes a complete measurement process on a 12-bit, 14-bit or 16-bit ADC.

### Syntax

```
ret_val = ADC(module, input_no)

#include ADwinPRO_ALL.inc
```

### Parameters

module	Specified module address (1...255).	LONG
input_no	number of the analog inputs (depending on the module: 1...8, 1...16 or 1...32).	LONG
ret_val	result of the conversion as 16-bit integer value (0...65535); at 12-bit ADCs the 4 LSBs are always 0, at 14-bit ADCs the 2 LSBs.	LONG

### Notes

The function **ADC** is characterized by a sequence of several commands:

SET_MUX	→	...	→	START_CONV	→	WAIT_EOC	→	READADC
Set the multiplexer to the specified input		Wait for settling of the multiplexer (3µs)		Start A/D conversion		Wait for end of conversion		Read out the converted value
with AIn-x/16 Rev. B or higher: 14µs								

In the following cases you should use the instructions mentioned above instead of the instruction **ADC**:

- Very short cycle times: **GLOBALDELAY** < 200 (see above).
- High internal resistance (>3kΩ) of the voltage source of the measurement signal: This increases the settling time of the multiplexer to more than 3.0µs or 14µs.
- You would like to use waiting times for additional program tasks.

In two parallel processes which have different priority you are not allowed to apply this function with the same A/D-module.

A process with low priority can be interrupted in the middle of an **ADC** sequence by a process with higher priority. When the process with higher priority sets the multiplexer to another channel during this interruption, the function of the process with low priority returns the measurement value from the wrong channel.

Several parallel processes with high priority can apply the function **ADC** without any problems, because processes with high priority do not interrupt each other.

If the modules Pro-AIn-32/x are processed with differential inputs (see **SE\_DIFF**), you can use the numbers 1...8 and 17...24 for **input\_no**. On these modules, the numbers 9...16 and 25...32 will be used with single ended inputs only.

### See also

**SET\_MUX**, **START\_CONV**, **WAIT\_EOC**, **READADC**, **ADC16**



## To be used for the modules

(LP)SH-8(-FI), Aln-16/14-C Rev. A, Aln-32/12 Rev. A, Aln-32/12 Rev. B, Aln-32/14 Rev. A, Aln-32/16 Rev. B, Aln-32/16 Rev. C, Aln-8/12 Rev. A, Aln-8/12 Rev. B, Aln-8/14 Rev. A, Aln-8/16 Rev. B, Aln-8/16 Rev. C

## Example

```
#INCLUDE ADwinPRO_ALL.inc
```

```
DIM value AS LONG
```

```
EVENT:
```

```
    value = ADC(1, 4)           'Measure a value at the analog input 4
```

## ADC16

**ADC16** executes a complete measurement on a 16-bit ADC.  
The information apply only for the module Pro-AIn-8/16 REVA.

## Syntax

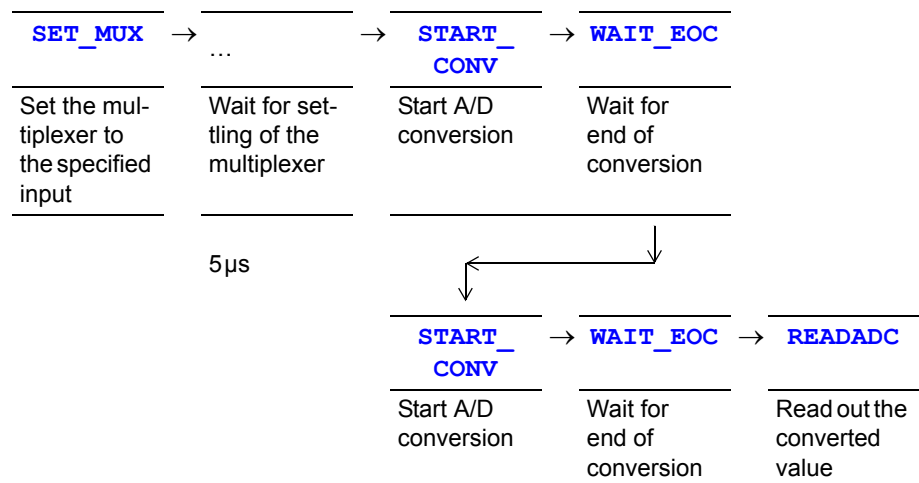
```
#INCLUDE ADwinPRO_ALL.inc  
ret_val = ADC16(module,input_no)
```

## Parameters

module	Specified module address (1...255).	LONG
input_no	Number of the analog input (1...8).	LONG
ret_val	Result of the conversion (0...65535).	LONG

## Notes

The function **ADC16** is characterized by a sequence of several instructions:



In the following cases you should use the instructions mentioned above instead of the instruction **ADC16**:

- Very short cycle times: **GLOBALDELAY** < 200 (see above).
- High internal resistance (>3kΩ) of the voltage source of the measurement signal: This increases the settling time of the multiplexer to more than 3.0µs or 14µs.
- You would like to use waiting times for additional program tasks.

## Start conversion twice

Starting the A/D conversion on this module results in the following 2 parallel reactions:

1. The current voltage value will be converted and saved in the temporary ADC register.
2. The value being in the temporary register before the conversion, is transferred serially from the ADC to the logic of the module and will be available after the end of conversion as return value.

The conversion has to be started twice so that the current value is presented by the ADC16-function and not the value of the last measurement.

In two parallel processes which have different priority you are not allowed to apply this function with the same A/D module.

A low-priority process can be interrupted in an **ADC16** sequence by a process with high priority. When the process with high priority sets the multiplexer to another channel, the function of the process with low priority may output the measurement value from the wrong channel.





Several parallel processes with high priority can apply the function **ADC16** without any problems, because processes with high priority do not interrupt each other.

## See also

[SET\\_MUX](#), [START\\_CONV](#), [WAIT\\_EOC](#), [READADC](#), [ADC](#)

## To be used for the modules

(LP)SH-8(-FI), AIn-8/16 Rev. A

## Example

```
#INCLUDE ADwinPRO_ALL.inc
```

```
DIM value AS LONG
```

```
EVENT:
```

```
    value = ADC16(1, 7)      'Measure a value at the analog input 7
```



## ADCF

**ADCF** executes a complete measurement on a Fast-ADC.

### Syntax

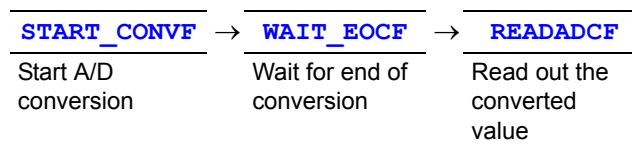
```
#INCLUDE ADwinPRO_ALL.inc
ret_val = ADCF(module, input_no)
```

### Parameters

module	Specified module address (1...255).	LONG
input_no	Number of the analog input (1...4 or 1...8).	LONG
ret_val	Result of the conversion (0...65535); on a 12-bit and 14-bit ADC the "missing" least significant bits are always set to 0.	LONG

### Notes

The function **ADCF** is characterized by a sequence of several instructions:



### See also

[START\\_CONV](#), [WAIT\\_EOCF](#), [READADCF](#), [READ\\_ADCF4](#), [READ\\_ADCF8](#), [READ\\_ADCF4\\_PACKED](#), [READ\\_ADCF8\\_PACKED](#)

### To be used for the modules

Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A

### Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG

EVENT:
  value = ADCF(1, 4)      'Measures a value at the analog input
4
```

**BURST\_ABORT** aborts a running burst-measurement sequence on the specified module.

The function returns the number of the measurements being already executed (= stored measurement values).

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
ret_val = BURST_ABORT(module)
```

## Parameters

module	Specified module address (1...255).	LONG
ret_val	Number of the stored measurement values (0...1048575 = $2^{20} - 1$ ).	LONG

## Notes

If you have aborted the burst-measurement sequence, the function **BURST\_STATUS** returns the number of the measurements not yet executed.

With the instruction **BURST\_READ** you can get already stored samples from the module after abort of the burst-measurement sequence.

## See also

[BURST\\_INIT](#), [BURST\\_START](#), [BURST\\_STATUS](#)

## To be used for the modules

Aln-F-4/14 Rev. B, Aln-F-8/14 Rev. B

## BURST\_ABORT

## Example

```
'Measurement with high priority process, reading out in the
'low-priority section FINISH:.
'Measurement will abort with a trigger signal at DIO32 module 1
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADWINPRO.INC
#INCLUDE ADwinPRO_ALL.inc

#DEFINE samples 10000      'Number of measurements to be
executed-+
#DEFINE ainadr 1           'Address AIn moduld
#DEFINE dioadr 1           'Address DIO module
#DEFINE sampleperiod 800   'Measurement rate of 50 kHz
                               '[=1/(25ns*800)]

DIM DATA_1[samples] AS LONG
DIM DATA_2[samples] AS LONG
DIM data_num AS LONG      'Number of converted measurement
values
DIM run_state AS LONG     'Sequence status of measurement:
                               'not running/running/finished
DIM cancel AS LONG        'Marker, if abort is requested

INIT:
    BURST_INIT(ainadr,1,sampleperiod,samples)
                               'Set address, mode, meas. rate,
                               'number of measurements
    run_state=0              'Set status: measurement sequence is
                               'not running
    DIGPROG1(dioadr,0)      'DIO 32: Bits 0...15 as input

EVENT:
    IF (run_state=0) THEN    'No measurement sequence running?
        BURST_START(ainadr)  'then start measurement sequence
        run_state=1          'Measurement sequence is running
    ENDIF
    IF (run_state=1) THEN    'If measurement sequence is running
        data_num=BURST_STATUS(ainadr) 'get number of the remaining
        'measurements
        cancel=DIGIN_WORD1(dioadr) 'External abort desired?
        IF (cancel AND 1=1) THEN 'Abort characteristic met?
            data_num=BURST_ABORT(ainadr) 'Abort measurement sequence /
            'number of measurement values
        END                  'End of program
    ENDIF
    IF (data_num=0) THEN END 'Are all measurements executed?:
        'terminate

ENDIF

FINISH: 'Get measurement values in low-priority section
    BURST_READ(ainadr,1,1,data_num,DATA_1,1)
```

**BURST\_CREAD** copies the measurement values of a channel, stored on the specified module, into an array. The number of measurement values to be copied has to be indicated.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
```

```
BURST_CREAD(module, channel, length, array[], arr_idx)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>channel</code>	Channel to be transferred (1...4; with a Pro-Aln-F-8/14 Rev. B: 1...8).	LONG
<code>length</code>	Number of the measurement values to be transferred (1...n), divided by 2.	LONG
<code>array[]</code>	Destination array into which the measurement values are transferred; no FIFO array allowed. .	ARRAY LONG
<code>arr_idx</code>	Destination startindex: Array element from which the storage of the measurement values is started (1...n).	LONG

## Notes

The measurement values are written into the lower word (bits 15...0) of an array element (a zero into the higher word).

The destination array must at least be dimensioned with `arr_idx + length` elements, in order to receive all measurement values.

Read out measurement values of a continuous burst-measurement sequence only with the instruction **BURST\_CREAD** (see **BURST\_CSTART**).



## See also

[BURST\\_INIT](#), [BURST\\_CSTART](#), [BURST\\_STATUS](#), [SET\\_GAIN](#), [SYNC\\_MODE](#)

## Can be also used for the modules

Pro-Aln-F-4/14 Rev. B, Pro-Aln-F-8/14 Rev. B

## BURST\_CREAD

### Example

```
#INCLUDE ADwinPRO_ALL.inc
#DEFINE length 10000
#DEFINE module 1
#DEFINE sampleperiod 20      '2000 kHz measurement rate
                              '[=1/(25ns*20)]

DIM DATA_1[length] AS LONG

INIT:
  'Address, mode, meas. rate, number of measurements
  BURST_INIT(module,1,sampleperiod,length)
  'Start continous burst-measurement
  BURST_CSTART(module)
  GLOBALDELAY=80             'Find the trigger point with 500 kHz

EVENT:
  PAR_1=READADCF(module,1) 'Get the latest measurement value
  IF (PAR_1>49152) THEN     'If +5V are exceeded
    PAR_9=BURST_ABORT(module) 'abort measurement and
    END                      'end process (= execute FINISH)
  ENDIF

FINISH:
  'Copy the last data into DATA_1
  BURST_CREAD(module,1,length,DATA_1,1)
```

**BURST\_CSTART** starts a burst-measurement sequence in the "Continuous" mode on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
BURST_CSTART(module)
```

## Parameters

module	Specified module address (1...255).	LONG
--------	-------------------------------------	------

## Notes

The instruction uses the built-in memory as ring buffer. During a continuous burst-measurement sequence measurements are executed in fixed time intervals and the values are stored in the ring buffer.

The stored measurement values are read out with **BURST\_CREAD** after the measurement sequence has finished (not with **BURST\_READ**).

As an example, this instruction enables a user to acquire (and to process) a number of measurement values directly before or after a trigger condition.

For this purpose the measurement is stopped by **BURST\_ABORT** immediately when (or a certain time interval after) the trigger condition occurred.



## See also

[BURST\\_INIT](#), [BURST\\_CREAD](#), [BURST\\_STATUS](#), [SET\\_GAIN](#), [SYNC\\_MODE](#)

## To be used for the modules

Aln-F-4/14 Rev. B, Aln-F-8/14 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc
#DEFINE length 10000
#DEFINE module 1
#DEFINE sampleperiod 20    '2000 kHz measurement rate
                           ' [=1/(25ns*20)]
DIM DATA_1[length] AS LONG

INIT:
  BURST_INIT(module,1,sampleperiod,length) 'Address, mode,
                                           'meas. rate, number of measurements
  BURST_CSTART(module)                  'Start continuous burst-measurement
  GLOBALDELAY=80                        'Find the trigger point with 500 kHz

EVENT:
  PAR_1=READADCF(module,1) 'Get the latest measurement value
  IF (PAR_1>49152) THEN    'If +5V is exceeded
    PAR_9=BURST_ABORT(module) 'abort measurement and
    END                      'end process (= execute FINISH)
  ENDIF

FINISH:
  'Copy the last data into DATA_1
  BURST_CREAD(module,1,length,DATA_1,1)
```

## BURST\_INIT

**BURST\_INIT** sets the parameters for a burst-measurement sequence on the specified module.

These are: Amount and number of the measurement channels, period duration of the measurement sequence and the number of the measurements to be carried out.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
BURST_INIT (module, mode, pulses, samples)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG																		
<code>mode</code>	Measurement mode (1...3; with a Pro-In-F-8/14 Rev. B: 1...4) determines the amount of measurement channels, the channel numbers are determined automatically. The measurement mode and the memory size determine the maximum amount of measurement values per channel:	LONG																		
<table> <tr> <th><code>mode</code></th><th>Channel no.</th><th>max. amount of measurement values per channel:</th></tr> <tr> <td>.</td><td>n.</td><td><math>1 \dots 2^{(n-1)}</math>      <math>2^{(21-n)} - 1</math></td></tr> <tr> <td>1.</td><td>1</td><td><math>2^{20} - 1 = 1048575 = 0FFFFFFH</math></td></tr> <tr> <td>2.</td><td>1, 2</td><td><math>2^{19} - 1 = 524287 = 7FFFFFFH</math></td></tr> <tr> <td>3.</td><td>1...4</td><td><math>2^{18} - 1 = 262143 = 3FFFFFFH</math></td></tr> <tr> <td>4.</td><td>1...8</td><td><math>2^{17} - 1 = 131071 = 1FFFFFFH</math></td></tr> </table>			<code>mode</code>	Channel no.	max. amount of measurement values per channel:	.	n.	$1 \dots 2^{(n-1)}$ $2^{(21-n)} - 1$	1.	1	$2^{20} - 1 = 1048575 = 0FFFFFFH$	2.	1, 2	$2^{19} - 1 = 524287 = 7FFFFFFH$	3.	1...4	$2^{18} - 1 = 262143 = 3FFFFFFH$	4.	1...8	$2^{17} - 1 = 131071 = 1FFFFFFH$
<code>mode</code>	Channel no.	max. amount of measurement values per channel:																		
.	n.	$1 \dots 2^{(n-1)}$ $2^{(21-n)} - 1$																		
1.	1	$2^{20} - 1 = 1048575 = 0FFFFFFH$																		
2.	1, 2	$2^{19} - 1 = 524287 = 7FFFFFFH$																		
3.	1...4	$2^{18} - 1 = 262143 = 3FFFFFFH$																		
4.	1...8	$2^{17} - 1 = 131071 = 1FFFFFFH$																		
<code>pulses</code>	determines the period duration of a measurement sequence as number of time intervals: period duration = <code>pulses</code> * 25ns (min. value = 20, is equivalent to 2MHz). The period duration is the time from the beginning of a measurement until the beginning of the next measurement.	LONG																		
<code>samples</code>	The amount of measurements per channel to be executed (the maximum value for <code>samples</code> is determined by <code>mode</code> ).	LONG																		

## Notes

Even if you do not use all measurement channels, all existing channels will always be converted during each measurement sequence. Selecting the measurement channels with **BURST\_INIT** only refers to the process of saving the converted measurement values.

You can even read with **READADCF** the current measurement value of a channel which is not saved, for instance for testing a trigger condition.

You can execute burst-measurement sequences on several modules synchronously, when you release the relevant modules with **SYNC\_MODE** for synchronization. If so, all measurements of the burst-measurement sequences can be carried out at the same time (synchronously). Please note, that the amount of the burst-measurements should be the same in the various burst-measurement sequences.



The instructions **ADCF** and **START\_CONV** will overwrite a setup done with **BURST\_INIT**, that means the measurement mode will be set to a value of 1. Therefore you must not use these instruction between **BURST\_INIT** and **BURST\_START** for safety purposes.



## See also

**BURST\_ABORT**, **BURST\_READ**, **BURST\_READ\_PACKED**, **BURST\_START**, **BURST\_STATUS**, **READADCF**, **SET\_GAIN**, **SYNC\_MODE**

## To be used for the modules

Aln-F-4/14 Rev. B, Aln-F-8/14 Rev. B

## Example

REM Measurement with high-priority process; as soon as a voltage REM higher than 5 V is measured, switch to burst-mode and measure REM with 1.0 MHz. Read out measurement values in the low-priority REM section FINISH:

```
#INCLUDE ADwinPRO_ALL.inc
#include ADWINPRO.INC
#define samples 10000      'Amount of measurements to be executed
#define ainadr 1           'Address of AIn module
#define sampleperiod 40    '1.0 MHz measurement rate
                           '=1/(25ns*40)

DIM DATA_1[samples] AS LONG
DIM DATA_2[samples] AS LONG
DIM dig_value AS LONG      'Supplied voltage value
DIM remaining AS LONG      'Amount of the remaining measurement
                           'values
DIM run_state AS LONG      'Status of the measurement sequence:
                           'not running/running/finished

INIT:
  run_state=0              'Set status: measurement sequence is
                           'not running

EVENT:
  IF (run_state=0) THEN    'Is no measurement sequence running?
    dig_value =ADCF(ainadr,1)
    IF (dig_value>49151) THEN 'Voltage >5 V
      'Address, mode, meas. rate, amount of measurements
      BURST_INIT(ainadr,2,sampleperiod,samples)
      BURST_START(ainadr) 'then start measurement sequence
      run_state=1         'Measurement sequence is running
    ENDIF
  ENDIF
  IF (run_state=1) THEN    'Is the measurement sequence running?
    remaining=BURST_STATUS(ainadr) 'Determining the remaining number
    'of measurements
    IF (remaining=0) THEN END 'Are all measurements executed?
  ENDIF

FINISH: 'Read out data in the low-priority section FINISH:
  BURST_READ(ainadr,1,1,samples,DATA_1,1) 'Read out measurement
    'values of channel 1
  BURST_READ(ainadr,2,1,samples,DATA_2,1) 'Read out measurement
    'values form channel 2
```

## BURST\_READ

**BURST\_READ** copies the measurement values of a channel into a specified array.

The amount of measurement values to be copied has to be indicated.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

BURST_READ (module, channel, startadr, count,
            array[], array_idx)
```

### Parameters

<code>module</code>	Specified module address (1...255)..	LONG
<code>channel</code>	Channel to be transferred (1...4; with a Pro-Aln-F-8/14 Rev. B: 1...8).	LONG
<code>startadr</code>	Source start address: Address from which it is started to read the measurement values.	LONG
<code>count</code>	Amount of the measurement values to be transferred (1...n).	LONG
<code>array[]</code>	Destination array, into which the measurement values are transferred; no FIFO array allowed.	ARRAY LONG
<code>array_idx</code>	Source startindex: First array element, where measurement values are stored (1...n).	LONG

### Notes

The measurement values are written into the lower word (bits 15...0) of an array element (a zero into the higher word).

The destination array must at least be dimensioned with `array_idx + length-1` elements, in order to receive all measurement values.

If you execute the instruction **BURST\_READ** during a running burst-measurement sequence, errors may occur. Therefore make sure that the burst-measurement sequence has finished. There are 2 possibilities:

- Check the status of the measurement sequence with the instruction **BURST\_STATUS**. The return value 0 (zero) shows that the measurement sequence has finished (otherwise you have to wait for the end of the measurement sequence).
- You abort the measurement sequence with **BURST\_ABORT**.

In a high-priority process you are only allowed to read as much data with **BURST\_READ** from the module, that the workload of the ADwin system does not exceed 100%. If this happens anyway, the communication with the computer may become instable (time-out).

In order to ensure a safe operation we recommend to use the instruction **BURST\_READ** only in a low-priority process or in a low-priority program section (e.g. **FINISH**).

### See also

[BURST\\_ABORT](#), [BURST\\_INIT](#), [BURST\\_READ\\_PACKED](#), [BURST\\_START](#), [BURST\\_STATUS](#), [SET\\_GAIN](#), [SYNC\\_MODE](#)

### To be used for the modules

Aln-F-4/14 Rev. B, Aln-F-8/14 Rev. B



## Example

```

REM Attention: Measurement in high-priority process, read out in
REM low-priority section!
REM Starts with positive edge at DIO bit 0 an analog measurement
REM at channel 1
#include ADwinPRO_ALL.inc
#include ADwinPRO_ALL.inc
#define samples 10000      'Number of measurements to be executed
#define sampleperiod 30    '1.33 MHz meas. rate [=1/(25ns*30)]
#define dioadr 1           'Module address DIO module
#define ainadr 1           'Module address AIN module
#define run_state PAR_1    'Status of the measurement sequence
                           'not running/running/finished

DIM DATA_1[samples] AS LONG
DIM remaining AS LONG      'Number of the remaining measurement
                           'values

DIM i AS LONG
DIM trigger AS LONG        'Marker for external trigger signal

INIT:
  DIGPROG1(dioadr,0)        'DIO bits 0...15 as inputs
  run_state=0              'Set status: measurement sequence is
                           'not running
  BURST_INIT(ainadr,1,sampleperiod,samples)
                           'Address, mode, meas. rate, amount of
                           'measurements

EVENT:
  IF (run_state=0) THEN     'no measurement sequence running?
    trigger=DIGIN_WORD1(dioadr)'read trigger signal
    IF (trigger AND 1=1) THEN 'trigger?
      BURST_START(ainadr)   'start measurement sequence
      run_state=1           'Measurement sequence is running
    ENDIF
  ENDIF
  IF (run_state=1) THEN     'If measurement sequence is running
    remaining=BURST_STATUS(ainadr)'Amount of remaining
                           'measurements
    IF (remaining=0) THEN END 'All measurements finished
                           ''... then terminate
  ENDIF

FINISH: 'Read out data in low-priority section FINISH:
  BURST_READ(ainadr,1,1,samples,DATA_1,1)
                           'Module address, channel, start addr.
                           'length, destination array,
                           'startindex in the destination array

```

## BURST\_READ\_PACKED

**BURST\_READ\_PACKED** copies the stored measurement values of a channel into a specified array. The values are packed and the copying process is effected quickly.

The amount of the measurement values which you want to copy have to be indicated.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

BURST_READ_PACKED (module, channel, startadr, count,
                   array[], array_idx)
```

### Parameters

module	Specified module address (1...255)..	LONG
channel	Channel to be transferred (1...4; with a Pro-AIn-F-8/14 Rev. B: 1...8).	LONG
startadr	Source start address: Address from which it is started to read the measurement values.	LONG
count	Amount of the measurement values to be transferred, divided by 2 (1...n/2).	LONG
array[]	Destination array, where the measurement values are transferred; no FIFO array allowed.	ARRAY LONG
array_idx	Destination startindex: Array element from which you start storing the measurement values (1...n).	LONG

### Notes

The measurement values are written alternately in the lower and upper word of an array element (see table). The destination array must at least be dimensioned by `array_idx + length` in order to get all measurement values.

Address in the destination <code>array[]</code>	Higher word (bits 31...16)	Lower word (bits 15...0)
<code>array_idx</code>	Meas. value 2	Meas. value 1
<code>array_idx + 1</code>	Meas. value 4	Meas. value 3
...	...	...
<code>array_idx + length</code>	Meas. value n	Meas. value (n-1)

If an odd amount of measurement values is stored on the module, **BURST\_READ\_PACKED** copies the last measurement value to the *lower* word, and a non-specified value to the higher word.

If you execute the instruction **BURST\_READ\_PACKED** during a running burst-measurement sequence, errors may occur. Therefore make sure that the burst-measurement sequence has finished. There are 2 possibilities:

- Check the sequence status of the measurement with the instruction **BURST\_STATUS**. The return value 0 (zero) shows that the measurement sequence has finished (otherwise you have to wait for the end of the measurement sequence).
- You abort the measurement sequence with **BURST\_ABORT**.

In a high-priority process you are only allowed to read as much data with **BURST\_READ\_PACKED** from the module, that the workload of the ADwin



system does not exceed 100%. If this happens anyway, the communication with the computer may become instable (time-out).

In order to ensure a safe operation we recommend to use the instruction **BURST\_READ\_PACKED** only in a low-priority process or in a low-priority program section (e.g. **FINISH**:).

## See also

**BURST\_ABORT**, **BURST\_INIT**, **BURST\_READ**, **BURST\_START**, **BURST\_STATUS**, **SET\_GAIN**, **SYNC\_MODE**

## To be used for the modules

Aln-F-4/14 Rev. B, Aln-F-8/14 Rev. B

## Example

REM Attention: Measurement in high-priority process, read out in  
REM the low-priority section!

```
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADWINPRO.INC
#DEFINE samples 10000      'Amount of measurements to be executed
#DEFINE ainadr 2           'Address of the AIN module
#DEFINE sampleperiod 40    '1 MHz measurement rate
                           '=1/(25ns*40)

DIM DATA_1[samples] AS LONG
DIM DATA_2[samples] AS LONG
DIM remaining AS LONG      'Amount of the remaining measurement
                           'values
DIM run_state AS LONG      'Status of the measurement sequence:
                           'not running/running/finished

INIT:
  BURST_INIT(ainadr,2,sampleperiod,samples)
                           'Address, mode, meas. rate, amount of
                           'measurements
  run_state=0              'Measurement sequence is not running
  SET_GAIN(ainadr,1,1)     'Measurement range channel 1 +/-5V
  SET_GAIN(ainadr,2,2)     'Measurement range channel 2 +/-2.5 V

EVENT:
  IF (run_state=0) THEN    'If no measurement sequence is running
    BURST_START(ainadr)    'start measurement sequence
    run_state=1            'Measurement sequence is running
  ENDIF
  IF (run_state=1) THEN    'If measurement is running
    remaining=BURST_STATUS(ainadr) 'Amount of remaining
    'measurements
    IF (remaining=0) THEN run_state=2 'If measurement sequence is
    'finished, set marker
  ENDIF

FINISH: 'Read out data in the low-priority section FINISH:
  BURST_READ_PACKED(ainadr,1,1,samples,DATA_1,1) 'Read
    'measurement values from channel 1
  BURST_READ_PACKED(ainadr,2,1,samples,DATA_2,1) 'Read
    'measurement values from channel 2
```

## BURST\_START

**BURST\_START** starts a burst-measurement sequence on the specified module (independent of the processor).

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
BURST_START(module)
```

### Parameters

module      Specified module address (1...255).

LONG

### Notes

Burst-measurement sequences can be synchronized. There are 2 possibilities:

1. *Start* burst-measurement sequence synchronously with other measurements:

The module must be released with **SYNCENABLE** for synchronization. The burst-measurement sequence is then started with the instruction **P2\_SYNCALL** synchronously with other measurements.

This mode synchronizes only the start, not the following execution; but synchronization with individual measurements is possible, too.

2. *Execute* measurements of several burst-measurement sequences synchronously:

The module must be released with **SYNC\_MODE** for synchronization (modes 2 and 3) and with **BURST\_START** for starting. The conversions of the measurement sequence are triggered by synchronization signals.

In master / slave mode (see **SYNC\_MODE**), release the burst-measurements on the slave modules first and then on the master module.

With event-controlled modules (see **SYNC\_MODE**), do first release all burst-measurement sequences with **BURST\_START** for starting and only then release the event inputs with **EVENTENABLE**.

### See also

[BURST\\_ABORT](#), [BURST\\_INIT](#), [BURST\\_READ](#), [BURST\\_READ\\_PACKED](#), [BURST\\_STATUS](#), [SET\\_GAIN](#), [SYNCENABLE](#), [SYNC\\_MODE](#)

### To be used for the modules

AIIn-F-4/14 Rev. B, AIIn-F-8/14 Rev. B



## Example

REM Measurement in a high-priority process; as soon as a voltage REM higher than 5 V is measured, switch to burst-mode and measure REM with 1.0 MHz. Read out measurement values in the low-priority REM section FINISH:

```
#INCLUDE ADwinPRO_ALL.inc
#include ADWINPRO.INC
#define samples 10000      'Amount of measurement being executed
#define ainadr 1           'Address of the AIn moduld
#define sampleperiod 40    '1 MHz measurement rate
                           '=1/(25ns*40)

DIM DATA_1[samples] AS LONG
DIM DATA_2[samples] AS LONG
DIM dig_value AS LONG     'Voltage value
DIM remaining AS LONG     'Amount of the remaining meas. values
DIM run_state AS LONG     'Status of the measurement sequence:
                           'not running/running/finished

INIT:
    run_state=0            'Set status: Measurement sequence is
                           'not running

EVENT:
    IF (run_state=0) THEN  'If no measurement sequence is running
        dig_value =ADCF(ainadr,1)
        IF (dig_value>49151) THEN 'voltage >5 V
            BURST_INIT(ainadr,2,sampleperiod,samples) 'Address, mode,
                                                         'measurement rate, amount of
                                                         'measurements
            BURST_START(ainadr) 'then start measurement sequence
            run_state=1         'Measurement sequence is running
        ENDIF
    ENDIF
    IF (run_state=1) THEN  'If measurement sequence is running
        remaining=BURST_STATUS(ainadr) 'determine the remaining amount
                                         'of measurements
        IF (remaining=0) THEN END 'All measurements finished
    ENDIF

FINISH: 'Read out data in the low-priority section FINISH:
    BURST_READ(ainadr,1,1,samples,DATA_1,1) 'Read out measurement
                                              'values from channel 1
    BURST_READ(ainadr,2,1,samples,DATA_2,1) 'Read out measurement
                                              'values from channel 2
```

## BURST\_STATUS

**BURST\_STATUS** determines the amount of the burst-measurements which are still to be executed on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
  
ret_val = BURST_STATUS(module)
```

### Parameters

module	Specified module address (1...255).	LONG
ret_val	Amount of measurements still to be executed.	LONG

### Notes

If a measurement sequence has already finished, the function returns the value 0 (zero).

### See also

[BURST\\_ABORT](#), [BURST\\_INIT](#), [BURST\\_CSTART](#), [BURST\\_CREAD](#),  
[BURST\\_READ](#), [BURST\\_READ\\_PACKED](#), [BURST\\_START](#), [SET\\_GAIN](#), [SYNC\\_MODE](#)

### To be used for the modules

Aln-F-4/14 Rev. B, Aln-F-8/14 Rev. B



## Example

```

REM Measurement in a high-priority process; as soon as a voltage
REM higher than 5 V is measured, switch to burst-mode and measure
REM with 1.0 MHz. Read out measurement values in the low-priority
REM section FINISH:
#include ADwinPRO_ALL.inc
#include ADWINPRO.INC
#define samples 10000 'Amount of the measurements being executed
#define ainadr 1 'Address of the AIn module
#define sampleperiod 40 '1 MHz measurement rate [=1/(25ns*40)]

DIM DATA_1[samples] AS LONG
DIM DATA_2[samples] AS LONG
DIM volt_value AS LONG 'Voltage value
DIM remaining AS LONG 'Amount of the remaining meas. values
DIM run_state AS LONG 'Status of the measurement sequence:
                        'not running/running/finished

INIT:
    run_state=0 'Set status: measurement sequence not
                'running

EVENT:
    IF (run_state=0) THEN 'If no measurement sequence is running
        volt_value =ADCF(ainadr,1)
        IF (volt_value>49151) THEN 'voltage >5 V
            BURST_INIT(ainadr,2,sampleperiod,samples) 'Address, mode,
                'measurement rate, amount of
                'measurements
            BURST_START(ainadr) 'then start measurement sequence
            run_state=1 'Measurement sequence is running
        ENDIF
    ENDIF
    IF (run_state=1) THEN 'If measurement sequence is running
        remaining=BURST_STATUS(ainadr) 'determine the remaining
            'measurements
        IF (remaining=0) THEN END 'All measurements finished
    ENDIF

FINISH: 'Read out data in the low-priority section FINISH:
    BURST_READ(ainadr,1,1,samples,DATA_1,1) 'Read out measurement
        'values from channel 1
    BURST_READ(ainadr,2,1,samples,DATA_2,1) 'Read out measurement
        'values from channel 2

```

## READADC

**READADC** reads the result of a conversion from the ADC register of the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = READADC(module)
```

### Parameters

module	Specified module address (1...255).	LONG
ret_val	Measurement value in the ADC register (0...65535).	LONG

### Notes

When working with a Pro-AIn-8/16 module, not the current measurement value, but the measurement value of the previous measurement is read out (see [ADC16](#)).

If the instruction is executed too early, that is during a conversion, the return value has a lower resolution.

### See also

[ADC](#), [ADC16](#), [READADC\\_SCONV](#), [SET\\_MUX](#), [START\\_CONV](#), [SYNCALL](#), [WAIT\\_EOC](#)

### To be used for the modules

(LP)SH-8(-FI), AIn-16/14-C Rev. A, AIn-32/12 Rev. A, AIn-32/12 Rev. B, AIn-32/14 Rev. A, AIn-32/16 Rev. B, AIn-32/16 Rev. C, AIn-8/12 Rev. A, AIn-8/12 Rev. B, AIn-8/14 Rev. A, AIn-8/16 Rev. A, AIn-8/16 Rev. B, AIn-8/16 Rev. C, AO-16/8-12

### Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM val1 AS LONG 'Declaration

EVENT:
    SET_MUX(1,0) 'Set multiplexer to input 1
                  'Wait 3µs (12-Bit-ADC) or
                  '14µs (AIn-8/16 Rev. B, AIn-32/16 Rev.
B)
                  'for the settling of the multiplexer*
    START_CONV(1) 'Start AD conversion
    WAIT_EOC(1)   'Wait for end of conversion
    val1 = READADC(1) 'Read value from the ADC
```

\*The waiting period can be bypassed for instance by some *ADbasic* instructions, which do not have access to the same A/D module that is responsible for changing the settling of the multiplexer.

If there is no necessity to use the waiting time for other instructions, the following loop can be programmed, which you are only allowed to use in high-priority processes (with an ADSP).

```
START_CONV(1) 'Start AD conversion
PAR_80 = READ_TIMER()
DO
UNTIL (READ_TIMER() - PAR_80 > 560) 'Wait 25ns*560=14µs
```

**READADC\_SCONV** reads out the conversion result from an ADC of the specified module and starts immediately a new conversion.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = READADC_SCONV(module)
```

## Parameters

module	Specified module address (1...255).	LONG
ret_val	Measurement value in the ADC register (0...65535).	LONG

## Notes

When using the module Pro-Aln-8/16 not the current measurement value but the value from the previous measurement is read out (see instruction [ADC16](#)).

If the instruction is executed too early, that is during a conversion, the return value has a lower resolution.

If you use the instruction **P2\_SYNCALL** you can no longer use **READADC\_SCONV**! Instead, use only the instruction **READADC**, because the conversion is started by **P2\_SYNCALL**.



## See also

[ADC](#), [ADC16](#), [READADC](#), [SE\\_DIFF](#), [SET\\_MUX](#), [START\\_CONV](#), [WAIT\\_EOC](#)

## To be used for the modules

(LP)SH-8(-FI), Aln-16/14-C Rev. A, Aln-32/12 Rev. B, Aln-32/14 Rev. A, Aln-32/16 Rev. B, Aln-32/16 Rev. C, Aln-8/12 Rev. B, Aln-8/14 Rev. A, Aln-8/16 Rev. A, Aln-8/16 Rev. B, Aln-8/16 Rev. C, AO-16/8-12

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM i AS LONG
DIM DATA_1[1000] AS LONG 'Declaration
```

### INIT:

```
i=1
SET_MUX(1,2) 'Set multiplexer to input 3
```

Wait 3µs (12-bit ADC) or 14µs (Aln-8/16 Rev. B, Aln-32/16 Rev. B) for the settling of the multiplexer.

```
START_CONV(1) 'Start A/D converter
```

### EVENT:

```
WAIT_EOC(1) 'Wait for end of conversion
DATA_1[i] = READADC_SCONV(1) 'Read out and start A/D converter
INC(i) 'Increment index
IF (i=1001) THEN END 'End process after 1000 measurement values
```

## READADCF

**READADCF** reads out the conversion result from an F-ADC of the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = READADCF(module, adc_no)
```

### Parameters

module	Specified module address (1...255).	LONG
adc_no	Number of the ADC being read (1...4 or 1...8).	LONG
ret_val	Measurement value in the F-ADC register (0...65535).	LONG

### Notes

With a 12-bit converter the 4 least-significant bits are always 0, with a 14-bit converter the 2 least-significant bits.

As an alternative, the instructions **READ\_ADCF4**, **READ\_ADCF8**, **READ\_ADCF4\_PACKED**, **READ\_ADCF8\_PACKED** are available, which read conversion results very fast.

### See also

[ADCF](#), [START\\_CONV](#), [WAIT\\_EOCF](#), [READADCF\\_32](#), [READADCF\\_SCONV](#), [READADCF\\_SCONV\\_32](#), [READ\\_ADCF4](#), [READ\\_ADCF8](#), [READ\\_ADCF4\\_PACKED](#), [READ\\_ADCF8\\_PACKED](#)

### To be used for the modules

Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A

### Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM val1 AS LONG 'Declaration

EVENT:
    START_CONV(1,1) 'Start AD conversion
    WAIT_EOCF(1,1) 'Wait for end of conversion
    val1 = READADCF(1,1) 'Read value from the ADC
```

**READ\_ADCF4** reads out the conversion results from the first 4 F-ADC of the specified module.

## Syntax

```
#INCLUDE ADWINPRO_ALL.INC

READ_ADCF4(module,array[],index)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>array[]</code>	Destination array, where conversion results are saved.	LONG
<code>index</code>	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 **READADCF**.

With a 12-bit converter the 4 least-significant bits are always 0, with a 14-bit converter the 2 least-significant bits.

## See also

[ADCF](#), [START\\_CONV](#), [READADCF](#), [READ\\_ADCF8](#), [READ\\_ADCF4\\_PACKED](#), [READ\\_ADCF8\\_PACKED](#), [READADCF\\_SCONV](#), [WAIT\\_EOCF](#)

## To be used for the modules

Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A

## Example

```
#INCLUDE ADWINPRO_ALL.INC
DIM value[4] AS LONG      'Array for conversion results

EVENT:
    START_CONV(1,0Fh)      'Start AD conversion channels 1...4

EVENT:
    WAIT_EOCF(1,1)         'Wait for end of conversion
    READ_ADCF4(1,value,1)  'Read values of ADC 1...4
    START_CONV(1,0Fh)      'Start new AD conversion
```

## READ\_ADCF4

## READ\_ADCF8

**READ\_ADCF8** reads out the conversion results from all 8 F-ADC of the specified module.

### Syntax

```
#INCLUDE ADWINPRO_ALL.INC
READ_ADCF8(module,array[],index)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>array[]</code>	Destination array, where conversion results are saved.	LONG
<code>index</code>	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 **READADCF**.

With a 12-bit converter the 4 least-significant bits are always 0, with a 14-bit converter the 2 least-significant bits.

### See also

[ADCF](#), [START\\_CONV](#), [WAIT\\_EOCF](#), [READ\\_ADCF4](#), [READ\\_ADCF4\\_PACKED](#), [READ\\_ADCF8\\_PACKED](#), [READADCF\\_SCONV](#)

### To be used for the modules

Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A

### Example

```
#INCLUDE ADWINPRO_ALL.INC
DIM value[8] AS LONG      'Array for conversion results

EVENT:
  START_CONV(1,0FFh)      'Start AD conversion channels 1...8

EVENT:
  WAIT_EOCF(1,1)          'Wait for end of conversion
  READ_ADCF8(1,value,1)   'Read values of ADC 1...8
  START_CONV(1,0FFh)      'Start new AD conversion
```

**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

READ_ADCF4_PACKED (module, array[], index)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>array[]</code>	Destination array, where conversion results are saved.	LONG
<code>index</code>	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

The instruction is faster than the use of **READ\_ADCF4**.

With a 12-bit converter the 4 least-significant bits are always 0, with a 14-bit converter the 2 least-significant bits.

## See also

[ADCF](#), [START\\_CONVF](#), [WAIT\\_EOCF](#), [READ\\_ADCF4](#), [READ\\_ADCF8](#), [READ\\_ADCF8\\_PACKED](#), [READADCF\\_SCONV](#)

## To be used for the modules

Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A

## Example

```
#INCLUDE ADWINPRO_ALL.INC
DIM value[2] AS LONG 'Array for conversion results

EVENT:
  START_CONVF (1,0Fh) 'Start AD conversion channels 1...4

EVENT:
  WAIT_EOCF (1,1) 'Wait for end of conversion
  READ_ADCF4_PACKED (1,value,1) 'Read values of ADC 1...4
  START_CONVF (1,0Fh) 'Start new AD conversion
```

## READ\_ADCF4\_PACKED

## READ\_ADCF8\_PACKED

**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

READ_ADCF8_PACKED (module, array[], index)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>array[]</code>	Destination array, where conversion results are saved.	LONG
<code>index</code>	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

The instruction is faster than the use of **READ\_ADCF8**.

With a 12-bit converter the 4 least-significant bits are always 0, with a 14-bit converter the 2 least-significant bits.

### See also

[ADCF](#), [START\\_CONV](#), [WAIT\\_EOCF](#), [READ\\_ADCF4](#), [READ\\_ADCF8](#), [READ\\_ADCF4\\_PACKED](#), [READ\\_ADCF8\\_PACKED](#), [READ\\_ADCF\\_SCONV](#)

### To be used for the modules

Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A

### Example

```
#INCLUDE ADWINPRO_ALL.INC
DIM value[4] AS LONG           'Array for conversion results

EVENT:
    START_CONV (1, 0FFh)       'Start AD conversion channels 1...8

EVENT:
    WAIT_EOCF (1, 1)           'Wait for end of conversion
    READ_ADCF8_PACKED (1, value, 1) 'Read values of ADC 1...8
    START_CONV (1, 0FFh)       'Start new AD conversion
```



**READADCF\_32** 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 = READADCF_32(module, adc_no)
```

## Parameters

module	Specified module address (1...255).	LONG
adc_no	Number of the first F-ADC to be read (1, 3 or 1, 3, 5, 7).	LONG
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.

With a 12-bit converter the 4 least-significant bits are always 0, with a 14-bit converter the 2 least-significant bits.

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

[ADCF](#), [READADCF](#), [READ\\_ADCF4](#), [READ\\_ADCF8](#), [READ\\_ADCF4\\_PACKED](#), [READ\\_ADCF8\\_PACKED](#), [READADCF\\_SCONV](#), [READADCF\\_SCONV\\_32](#), [START\\_CONVF](#), [WAIT\\_EOCF](#)

## To be used for the modules

Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM val1 AS LONG 'Declaration

EVENT:
  START_CONVF(1,3) 'Start AD conversion on ADC1 and ADC2
  WAIT_EOCF(1,3) 'Wait for the end of the conversions
  val1 = READADCF_32(1,1) 'Read value of ADC1 and ADC2
```

## READADCF\_32

## READADCF\_SCONV

**READADCF\_SCONV** reads out the conversion result from an F-ADC of the specified module and starts immediately a new conversion.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = READADCF_SCONV(module, adc_no)
```

### Parameters

module	Specified module address (1...255).	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

### Example

With a 12-bit converter the 4 least-significant bits are always 0, with a 14-bit converter the 2 least-significant bits.

### See also

[ADCF](#), [READADCF](#), [READ\\_ADCF4](#), [READ\\_ADCF8](#), [READ\\_ADCF4\\_PACKED](#), [READ\\_ADCF8\\_PACKED](#), [READADCF\\_32](#), [READADCF\\_SCONV\\_32](#), [START\\_CONVF](#), [WAIT\\_EOCF](#)

### To be used for the modules

Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A

### Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM i AS LONG
DIM DATA_1[1000] AS LONG 'Declaration

INIT:
  i=1
  START_CONVF(1,1) 'Start A/D converter

EVENT:
  WAIT_EOCF(1,1) 'Wait for end of conversion
  DATA_1[i] = READADCF_SCONV(1,1) 'Read out + start A/D converter
  INC(i) 'Increment index
  IF (i=1001) THEN END 'End process after 1000 measurement values
```

**READADCF\_SCONV\_32** reads the conversion results from the 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 = READADCF_SCONV_32(module, adc_no)
```

## Parameters

module	Specified module address (1...255).	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										

## Notes

With a 12-bit converter the 4 least-significant bits are always 0, with a 14-bit converter the 2 least-significant bits.

## See also

[ADCF](#), [READADCF](#), [READ\\_ADCF4](#), [READ\\_ADCF8](#), [READ\\_ADCF4\\_PACKED](#), [READ\\_ADCF8\\_PACKED](#), [READADCF\\_32](#), [READADCF\\_SCONV](#), [START\\_CONVF](#), [WAIT\\_EOCF](#)

## To be used for the modules

Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG           'Declaration

INIT:
    START_CONVF(1,3)         'Start AD conversion

EVENT:
    WAIT_EOCF(1,3)           'Wait for end of conversion
    value = READADCF_SCONV_32(1,1) 'Read value from ADC1 and ADC2
                                   'and start conversion of both ADCs
```

## READADCF\_SCONV\_32

## SE\_DIFF

**SE\_DIFF** sets the operating mode single ended or differential for all analog inputs on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
SE_DIFF(module,choice)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>choice</code>	Operating mode of the analog inputs. 0: single ended. 1: differential (default).	LONG

### Notes

In the operating mode single ended 32 inputs are available, in the operating mode differential 16 inputs. After power up all inputs are in the differential mode.



Pay attention to the different pin assignment for the configurations (see hardware documentation of the module).  
In differential operation the analog inputs are accessed only with numbers 1...8 and 17...24 .

### See also

[ADC](#)

### To be used for the modules

Aln-32/12 Rev. A, Aln-32/12 Rev. B, Aln-32/14 Rev. A, Aln-32/16 Rev. B, Aln-32/16 Rev. C

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
SE_DIFF(1,0)           'Module with the address 1
                        'is set to SE
SE_DIFF(2,1)           'Module with the address 2
                        'is set to DIFF
```

**SET\_GAIN** sets the operating mode for a channel of the specified module, and thus the gain factor and measurement range, too.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
SET_GAIN(module, channel, mode)
```

## Parameters

module	Specified module address (1...255).	LONG
channel	Channel, whose gain is set (1...4 or 1...8).	LONG
mode	Operating mode (0...3) of the channel: Determines the gain of the input signal. With the gain, the measurement range for the input signals changes inversely.	LONG

Operating mode mode	Gain 2 <sup>n</sup>	Measurement range ±10V / 2 <sup>n</sup>
0	1	±10 V
1	2	±5 V
2	4	±2.5 V
3	8	±1.25 V

## See also

[ADCF](#), [BURST\\_CSTART](#), [BURST\\_START](#), [READADCF](#), [START\\_CONV](#), [WAIT\\_EOCF](#)

## To be used for the modules

Aln-F-4/14 Rev. B, Aln-F-8/14 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc
#define ainadr 1 'Module address AIN module

INIT:
    SET_GAIN(ainadr, 4, 1)    'Set voltage range in channel 4
                              'to operating mode 1
                              '(measurement range: +5V...-5V)

EVENT:
    PAR_1 = ADCF(1, 4)        'Measures a value at the analog input
4
```

## SET\_GAIN

## SET\_MUX

**SET\_MUX** sets the multiplexer input of the module to a specified channel and gain.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
```

```
SET_MUX(module, pattern)
```

## Parameters

**module** Specified module address (1...255). LONG

**pattern** Bit pattern for setting the multiplexer (see table); 2 bits are setting the gain, 5 bits the number of the channel. LONG

Bit no.	31:7	6	5	4	3	2	1	0
Meaning	–	Gain		Multiplexer input				
		1 = 00b		Input 1: 00000b				
		2 = 01b		Input 2: 00001b				
		4 = 10b		Input 3: 00010b				
		8 = 11b		...				
				Input 32: 11111b				

## Notes

For setting the multiplexer, combine the relevant bit combination for gain and multiplexer input. You can use the bits in the parameter **pattern** in binary notation or convert them into hexadecimal or decimal format. Consider the additional letters **h** and **b** for hexadecimal and binary code.

Pay attention to the necessary settling time of the multiplexer (3µs with 12-bit ADCs, 14µs for Aln-8/16 Rev. B, Aln-32/16 Rev. B). Make sure that at least these time intervals pass between a new setting of the multiplexer and the start of conversion.

At a Pro-Aln-8/16 module the gain can be set. Here the bits 5 and 6 have no function.

## See also

[ADC](#), [ADC16](#), [START\\_CONV](#), [WAIT\\_EOC](#), [READADC](#)

## To be used for the modules

(LP)SH-8(-FI), Aln-16/14-C Rev. A, Aln-32/12 Rev. A, Aln-32/12 Rev. B, Aln-32/14 Rev. A, Aln-32/16 Rev. B, Aln-32/16 Rev. C, Aln-8/12 Rev. A, Aln-8/12 Rev. B, Aln-8/14 Rev. A, Aln-8/16 Rev. A, Aln-8/16 Rev. B, Aln-8/16 Rev. C, AO-16/8-12

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM val1 AS LONG          'Declaration

EVENT:
    SET_MUX(1,0100010b)    'Set MUX to input 3, set gain 2
                           'Wait 3µs (12-bit ADC) or
                           '14µs (AIn-8/16 Rev. B, AIn-32/16 Rev.
B)                          'for the settling of the multiplexer

    START_CONV(1)          'Start AD conversion
    WAIT_EOC(1)            'Wait for end of conversion
    val1 = READADC(1)      'Read value from the ADC
```

## SEQ\_MODE

**SEQ\_MODE** initializes the specified module for an operation with sequential control. The operating mode and the gain factor are set (the same for all channels).

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
SEQ_MODE(module, mode, gain)
```

## Parameters

module	Specified module address (1...255).	LONG
mode	Operating mode of the sequential control on the module: 0: Normal mode (default). 1: Mode "single shot". 3: Mode "continuous".	LONG
gain	Gain factor (for the modes 1 and 3 only): 0: Factor = 1. 1: Factor = 2. 2: Factor = 4. 3: Factor = 8.	LONG

## Notes

The modes 1 and 3 activate the sequential control of the module. This sequential control enables a user to execute with one instruction a conversion at several channels consecutively. The control is always related to those channels being selected by **SEQ\_SELECT**.

The differences between the modes are as follows:

Mode	Type of measurement
0 Normal:	Standard: Individual measurement at one channel (see <a href="#">ADC</a> ).
1 "single shot":	The sequential control is finished as soon as each of the selected channels is converted once.
3 "continuous":	The sequential control continuously converts new measurement values at the selected channels.

Up to 32 measurement values can be stored in the built-in module memory.

14-bit converters return the measurement result left-aligned in one 16-bit value, that is, the 2 least-significant bits are always zero.

If the internal resistance of the voltage source of the measurement signal is too high ( $>3k\Omega$ ), the predefined settling time of the multiplexer will not be sufficient for an exact measurement. You can change the settling time of the multiplexer with the instruction **SEQ\_SET\_DELAY**.

## See also

[SEQ\\_SELECT](#), [SEQ\\_SET\\_DELAY](#), [SEQ\\_STATUS](#), [SEQ\\_READ](#)

## To be used for the modules

Aln-16/14-C Rev. A, Aln-32/14 Rev. A, Aln-32/16 Rev. C, Aln-8/14 Rev. A, Aln-8/16 Rev. C





## Example

```
#DEFINE module 1
#include ADwinPRO_ALL.inc

DIM DATA_1[16] AS LONG AT DM_LOCAL

INIT:
    SE_DIFF(module,0)          'Set inputs to single ended
    SEQ_MODE(module,3,0)       'Sequential control: Continuous Mode,
                                'Gain factor 1
    SEQ_SELECT(module,55555555h) 'Measure all odd-numbered
                                'channels of the module AIN-32/..
    START_CONV(1)              'Start measurement sequence
                                '(Continuous Mode)
    WAIT_EOC(1)                'Wait, until all indicated channels
                                'are measured once

EVENT:
    SEQ_READ(module,16,DATA_1,1) 'Get measurement values from the
                                'module and copy them to DATA_1
```

## SEQ\_READ

**SEQ\_READ** copies a specified amount of measurement values (16-bit each) from the module to a destination array.

1 measurement value is copied into each of the array elements.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

SEQ_READ(module, length, array[], array_idx)
```

### Parameters

module	Specified module address (1...255).	LONG
length	Amount of the measurement values being read (1...32).	LONG
array[]	Destination array where the measurement values are transferred.	ARRAY LONG
array_idx	Destination startindex: Array element from which you start storing the measurement values (1...n).	LONG

### Notes

You can only reasonably use this instruction if the sequential control of the module has been activated before by **SEQ\_MODE**.

The measurement values of the measurement group (see **SEQ\_SELECT**) are copied into the destination array in ascending order beginning at the lowest channel number.

### See also

[SEQ\\_READ\\_ONE](#), [SEQ\\_READ\\_TWO](#), [SEQ\\_READ\\_PACKED](#), [SEQ\\_READ32](#), [SEQ\\_MODE](#), [SEQ\\_READ](#)

### To be used for the modules

Aln-16/14-C Rev. A, Aln-32/14 Rev. A, Aln-32/16 Rev. C, Aln-8/14 Rev. A, Aln-8/16 Rev. C

### Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[16] AS LONG AT DM_LOCAL

INIT:
    SE_DIFF(1,0)           'Single ended inputs
    SEQ_MODE(1,3,0)        'Sequential control to Continuous
Mode,
    SEQ_SELECT(1,55555555h) 'Gain factor 1
                           'Measure all odd-numbered channels
                           'of an AIN-32/.. module
    START_CONV(1)          'Start measurement sequences in the
                           'Continuous Mode
    WAIT_EOC(1)            'Wait, until all indicated channels
are
                           'measured once.

EVENT:
    SEQ_READ(1,16,DATA_1,1) 'Copy current measurement values from
                           'the module into DATA_1
```

**SEQ\_READ\_ONE** reads out a specified measurement value (16 bit) of a measurement group on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = SEQ_READ_ONE(module,idxno)
```

## Parameters

module	Specified module address (1...255).	LONG
idxno	Index no., that indicates the position of a channel in the measurement group (1 ... 32).	LONG
ret_val	The last saved measurement value of the channel with the position idxno in the measurement group.	LONG

## Notes

You can only reasonably use this instruction if the sequential control of the module has been activated before by **SEQ\_MODE**.

The index number must not be interpreted as the number of a channel, but it is the position number of a channel in the measurement group that was defined before by **SEQ\_SELECT**.

For instance you select with the index number 4 the channel 11 in a measurement group with the channels (1, 3, 7, 11, 12, 15).

## See also

[SEQ\\_MODE](#), [SEQ\\_SELECT](#), [SEQ\\_STATUS](#), [SEQ\\_READ](#)

## To be used for the modules

Aln-16/14-C Rev. A, Aln-32/14 Rev. A, Aln-32/16 Rev. C, Aln-8/14 Rev. A, Aln-8/16 Rev. C

## Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[32] AS FLOAT AT DM_LOCAL
DIM i AS LONG

INIT:
    SE_DIFF(1,0)           'Single ended inputs
    SEQ_MODE(1,1,0)        'Sequential control to Single Shot,
                           'Gain factor 1
    SEQ_SELECT(1,0FFFFFFFh) 'Measure all inputs 1...32 of an
                           'AIN-32/.. module

EVENT:
    START_CONV(1)          'Start measurement sequence in the
mode                        mode
                           'Single Shot
    FOR i=1 TO 32          'Read all 32 channels of an AIN-32/..
                           'module ...
    DO                     'as soon as the ...
    UNTIL(i<=SEQ_STATUS(1)) 'individual measurement has finished
    DATA_1[i]=(SEQ_READ_ONE(1,i)-32768)*20/65536
                           'convert digits into Volt and save the
                           'values
    NEXT i
```

## SEQ\_READ\_ONE

## SEQ\_READ\_TWO

**SEQ\_READ\_TWO** reads out at the same time 2 consecutive measurement values (16-bit each) of a measurement group, on the specified module and returns them in a 32-bit value.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
ret_val = SEQ_READ_TWO(module,idxno)
```

## Parameters

**module** Specified module address (1...255). LONG  
**idxno** Index number that indicates the position of 2 channels in the measurement group (0 ... 15). LONG

Indexno.	0	1	2	...	15
Position	1, 2	3, 4	5, 6	...	31, 32

**ret\_val** 32-bit value that contains the 2 measurement values of the channels (indirectly selected by **idxno**). LONG

## Notes

You can only reasonably use this instruction if the sequential control of the module has been activated before by **SEQ\_MODE**.

The index number must not be interpreted as the number of a channel, but it indicates the position of two channels in the measurement group defined by **SEQ\_SELECT**.

The returned 32-bit value contains in the lower word the measurement value of the channel with the lower of the two channel numbers, in the higher word you will find the values of the channel with the higher number.

For instance, the index number 1 in a measurement group with the channels (1, 3, 7, 11, 12, 15) means that the channels 7 and 11 are read out. The measurement value of channel 7 is returned in the lower word of the return value, the value of channel 11 in the higher word.

## See also

[SEQ\\_MODE](#), [SEQ\\_SELECT](#), [SEQ\\_STATUS](#), [SEQ\\_READ](#)

## Can be used for the modules

Pro-AIn-8/14 Rev. A, Pro-AIn-32/14 Rev. A

Pro-AIn-8/16 Rev. C, Pro-AIn-32/16 Rev. C

## Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[32] AS LONG AT DM_LOCAL
DIM i, value32 AS LONG

INIT:
    SE_DIFF(1,0)           'Single ended inputs
    SEQ_MODE(1,1,0)        'Sequential control to Single Shot,
                           'Gain factor 1
    SEQ_SELECT(1,0FFFFFFFh) 'Measure all 32 inputs of the AIN-
32/..

EVENT:
    START_CONV(1)          'Start measurement sequence in the
mode
                           'Single Shot
    FOR i=1 TO 15 STEP 2   'Read all 32 channels
        DO                'As soon as ...
            UNTIL ( (i+1)<=SEQ_STATUS(1) ) '2 measurements have finished:
                value32=SEQ_READ_TWO(1,(i-1)/2) 'get LONG word and save it
                DATA_1[i]=value32 AND 0FFFFh 'Save lower word
                DATA_1[i+1]=SHIFT_RIGHT(value32,16) 'Save higher word
            NEXT i
        WAIT_EOC(1)
```

## SEQ\_READ\_PACKED

**SEQ\_READ\_PACKED** copies an even amount of measurement values (16-bit each) in pairs from the specified module to a destination array.

2 measurement values are copied into each of the array elements.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
```

```
SEQ_READ_PACKED(module, length, array[], array_idx)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>length</code>	Amount of the measurement value pairs to read (1...16).	LONG
<code>array[]</code>	Destination array where the measurement value pairs are transferred.	ARRAY LONG
<code>array_idx</code>	Destination startindex: Array element from which you start storing the measurement values (1...n).	LONG

### Notes

You can only reasonably use this instruction if the sequential control of the module has been activated before by **SEQ\_MODE**.

The measurement values of the measurement group (see **SEQ\_SELECT**) are copied in pairs into the destination array in ascending order beginning at the lowest channel number. An array element contains in the lower word the measurement value of the channel with the lower of the two channel numbers, in the higher word you will find the values of the channel with the higher number.

### See also

[SEQ\\_READ\\_ONE](#), [SEQ\\_READ\\_TWO](#), [SEQ\\_READ\\_PACKED](#), [SEQ\\_READ32](#), [SEQ\\_MODE](#), [SEQ\\_READ](#)

### To be used for the modules

Aln-16/14-C Rev. A, Aln-32/14 Rev. A, Aln-32/16 Rev. C, Aln-8/14 Rev. A, Aln-8/16 Rev. C

## Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[32] AS LONG AT DM_LOCAL

INIT:
    SE_DIFF(1,0)           'Single ended inputs
    SEQ_MODE(1,3,0)        'Sequential control to Continuous
Mode,
                           'Gain factor 1
    SEQ_SELECT(1,0AAAAAAAh) 'Measure all even-numbered channels
of
                           'the AIN-32/.. module
    START_CONV(1)          'Start measurement sequences in the
                           'Continuous Mode
    WAIT_EOC(1)            'Wait, until all indicated channels
are
                           'measured once

EVENT:
    SEQ_READ_PACKED(1,8,DATA_1,1)
                           'Get 16 measurement values from the
                           'module and copy them to DATA_1
```

## SEQ\_READ32

**SEQ\_READ32** copies all 32 measurement values (16-bit each) from the specified module into the destination array.

1 measurement value is copied into each of the array element.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

SEQ_READ32(module, array[], array_idx)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>array[]</code>	Destination array where the measurement values are transferred.	ARRAY LONG
<code>array_idx</code>	Destination startindex: Array element from which you start storing the measurement values (1...n).	LONG

### Notes

You can only reasonably use this instruction if the sequential control of the module has been activated before by **SEQ\_MODE**.

The measurement values of the measurement group (see **SEQ\_SELECT**) are copied into the destination array in ascending order beginning at the lowest channel numbers.

If you need more than sixteen measurement values, this instruction is faster than **SEQ\_READ**, although **SEQ\_READ32** always transfers all 32 measurement values.

### See also

[SEQ\\_READ\\_ONE](#), [SEQ\\_READ\\_TWO](#), [SEQ\\_READ\\_PACKED](#), [SEQ\\_READ32](#), [SEQ\\_MODE](#), [SEQ\\_READ](#)

### To be used for the modules

Aln-16/14-C Rev. A, Aln-32/14 Rev. A, Aln-32/16 Rev. C, Aln-8/14 Rev. A, Aln-8/16 Rev. C

### Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[32] AS LONG AT DM_LOCAL

INIT:
    SE_DIFF(1,0)           'Single ended inputs
    SEQ_MODE(1,3,0)        'Sequential control to Continuous
Mode,
                           'Gain factor1
    SEQ_SELECT(1,0FFFFFFFh) 'Measure all channels with an AIN-
32/..
    START_CONV(1)          'Start measurement sequences in the
                           'Continuous Mode
    WAIT_EOC(1)            'Wait, until all indicated channels
are
                           'measured once

EVENT:
    SEQ_READ32(1,DATA_1,1) 'Get measurement values from the
                           'module and copy them to DATA_1
```



**SEQ\_SELECT** determines the channels belonging to the measurement group, which will be converted by the sequential control on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
SEQ_SELECT(module, pattern)
```

## Parameters

<b>module</b>	Specified module address (1...255).	LONG
<b>pattern</b>	Bit pattern that determines the channels, whose values are to be read.	LONG

Bit no.	31	...	8	...	2	1	0
Channel no.	32	...	9	...	3	2	1

## Notes

In the measurement group you can combine any of the 32 channels of the module. The channels of a measurement group are automatically sorted in ascending order of the channel numbers, that is, the sequential control converts the channel with the lowest number first.

The status and reading instructions are always related to the group of the selected channels.

At modules with 32 channels the inputs have to be set with **SE\_DIFF** to single ended or differential. Pay attention to the special numbering of the differential inputs (1 ... 8 and 17 ... 24).



## See also

[SE\\_DIFF](#), [SEQ\\_MODE](#), [SEQ\\_STATUS](#), [SEQ\\_READ](#)

## To be used for the modules

Aln-16/14-C Rev. A, Aln-32/14 Rev. A, Aln-32/16 Rev. C, Aln-8/14 Rev. A, Aln-8/16 Rev. C

## Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[32] AS LONG AT DM_LOCAL

INIT:
    SE_DIFF(1,1)           'Differential inputs
    SEQ_MODE(1,1,0)        'Sequential control to Single Shot,
                           'Gain factor 1
    SEQ_SELECT(1,0FF00FFh) 'Measure diff. inputs 1-8 and 17-24 of
                           'the module AIN-32/..

EVENT:
    START_CONV(1)          'Start measurement sequence in the
mode                       'mode
    ...                   'Single Shot
                           'Here the waiting time could be used
                           'reasonably
    WAIT_EOC(1)            'Wait until all indicated channels are
                           'measured once
    SEQ_READ(1,16,DATA_1,1) 'Get measurement values from the
                           'module and copy them to DATA_1
```

## SEQ\_SET\_DELAY

**SEQ\_SET\_DELAY** determines the settling time (waiting time between 2 measurements) of the sequential control on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
SEQ_SET_DELAY(module,time)
```

### Parameters

<b>module</b>	Specified module address (1...255).	LONG
<b>time</b>	Amount of time units. The settling time of the sequential control results from these time units. 0: Standard waiting time (default). 1...2047: Waiting time in time units of 25ns.	LONG

Bit no.	31	...	8	...	2	1	0
Channel no.	32	...	9	...	3	2	1

### Notes



Setting the settling time influences to a large extent the measurement results. Shorter settling times make more unprecise measurement results and longer settling times more precise results.

The settling time is calculated according to the following formula:

$$\text{Settling time} = \text{time} \cdot 25\text{ns} + \text{Conversion time}$$

You will find the values for the conversion time and the default setting of the settling time in the hardware documentation of your Pro module.

### See also

[SEQ\\_MODE](#), [SEQ\\_READ](#)

### To be used for the modules

Aln-16/14-C Rev. A, Aln-32/14 Rev. A, Aln-32/16 Rev. C, Aln-8/14 Rev. A, Aln-8/16 Rev. C

## Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[32] AS LONG AT DM_LOCAL

INIT:
    SE_DIFF(1,1)           'Differential inputs
    SEQ_MODE(1,1,0)        'Sequential control to Single Shot
    SEQ_SELECT(1,0FF00FFh) 'Measure diff. inputs 1-8 and 17-24 of
                           'the AIN-32/.. module
    SEQ_SET_DELAY(1,200)   'Allow the analog board to settle in
the                         'time of tconv + 200 * 25ns

EVENT:
    START_CONV(1)          'Start measurement sequence in the
mode                        '
                           'Single Shot
    ...                   'Here you could use the waiting time
                           'reasonably
    WAIT_EOC(1)            'Wait, until all indicated channels
are                          '
                           'measured once
    SEQ_READ(1,16,DATA_1,1) 'Get measurement values from the
                           'module and copy them to DATA_1
```

## SEQ\_STATUS

**SEQ\_STATUS** determines how many channels of the measurement group are already converted and stored by the sequential control of the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = SEQ_STATUS(module)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>ret_val</code>	Amount of the channels being already converted and stored (1 ... 32).	LONG

### Notes

It makes only sense to use this instruction in the mode 1 (single shot).

The return value must not be interpreted as the number of a channel, but always refers to the measurement group, defined by **SEQ\_SELECT**.

For instance, return value 4 in a measurement group with the channels (1, 3, 7, 11, 12, 15) means that channel 11 has already been converted and that channel 12 is waiting to be measured at next. In other words: There are already 4 values in the temporary register of the module waiting to be fetched. You can now either read out the values and process them or wait for the end of the measurement sequence.

### See also

[SEQ\\_MODE](#), [SEQ\\_SELECT](#), [SEQ\\_STATUS](#), [SEQ\\_READ](#)

### To be used for the modules

Aln-16/14-C Rev. A, Aln-32/14 Rev. A, Aln-32/16 Rev. C, Aln-8/14 Rev. A, Aln-8/16 Rev. C

### Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[32] AS FLOAT AT DM_LOCAL
DIM i AS LONG

INIT:
    SE_DIFF(1,0)           'Single ended inputs
    SEQ_MODE(1,1,0)        'Sequential control to Single Shot,
                           'Gain factor 1
    SEQ_SELECT(1,0FFFFFFFh) 'Measure all 32 inputs of the AIN-
32/..

EVENT:
    START_CONV(1)          'Start measurement sequence in the
mode
                           'Single Shot
    FOR i=1 TO 32          'Read all 32 channels ...
    DO                     'as soon as ...
        UNTIL(i<=SEQ_STATUS(1)) 'the measurement has finished ...
        DATA_1[i]=(SEQ_READ_ONE(1,i)-32768)*20/65536
                           'convert digits into Volt and save the
                           'result
    NEXT i
```

**SH\_SETMODE** sets the mode of the sample and hold levels.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
SH_SETMODE(module,mode)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>mode</code>	Mode of the sample & hold levels: 0: Sample. 1: Hold.	LONG

## Notes

In the mode "sample" the output voltage of the module follows the input voltage, whereas in the mode "hold" the output voltage is held at the same level as the value of the input voltage.

The output voltage can only be held on a constant level in a limited period of time (mode "hold"). The voltage drop (droop rate) is characteristically 1  $\mu$ V/ms at 25°C operating temperature, the acquisition time to 0.01% is approx. 20  $\mu$ s.

## See also

-/-

## To be used for the modules

(LP)SH-8(-FI)

## Example

```
#INCLUDE ADwinPRO_ALL.inc
```

### EVENT:

```
SH_SETMODE(1,0)      'Module with the address 1 is set
                      'to "sample" mode
SH_SETMODE(2,1)      'Module with the address 2 is set
                      'to "hold" mode
```

## SH\_SETMODE

## START\_CONV

**START\_CONV** starts the A/D conversion on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
START_CONV(module)
```

### Parameters

`module` Specified module address (1...255).

LONG

### Notes

If the module is released with **SYNCENABLE** for synchronization, the instruction **P2\_SYNCALL** has the same function as in **START\_CONV**.

### See also

[ADC](#), [ADC16](#), [READADC](#), [SE\\_DIFF](#), [SET\\_MUX](#), [SYNCALL](#), [WAIT\\_EOC](#)

### To be used for the modules

(LP)SH-8(-FI), Aln-16/14-C Rev. A, Aln-32/12 Rev. A, Aln-32/12 Rev. B, Aln-32/14 Rev. A, Aln-32/16 Rev. B, Aln-32/16 Rev. C, Aln-8/12 Rev. A, Aln-8/12 Rev. B, Aln-8/14 Rev. A, Aln-8/16 Rev. A, Aln-8/16 Rev. B, Aln-8/16 Rev. C, AO-16/8-12

### Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM value1 AS LONG 'Declaration
```

#### EVENT:

```
SET_MUX(1,0) 'Set multiplexer to input 1
```

Wait 3µs (12-bit ADC) or 14µs (Aln-8/16 Rev. B, Aln-32/16 Rev. B) for the settling of the multiplexer\*

```
START_CONV(1) 'Start AD conversion
WAIT_EOC(1) 'Wait for end of conversion
value1 = READADC(1) 'Read value from the ADC
```

\* The waiting period can be bypassed for instance by some *ADbasic* instructions, which do not have access to the same A/D module that is responsible for changing the settling of the multiplexer.

Another opportunity to bypass offers the following loop, which can only be used in processes with high priority:

```
time = READ_TIMER() 'Determine current timer rate
DO
UNTIL (READ_TIMER()-time>120) 'Wait 25ns*120=3µs
>560) 'Wait 25ns*560=14µs
```

Use for the Aln-8/16 Rev. B, Aln-32/16 Rev. B respectively

```
UNTIL (READ_TIMER()-time>560) 'Wait 25ns*560=14µs
```

**START\_CONV** starts the conversion of one or more F-ADCs of the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
START_CONV (module, adc_no)
```

## Parameters

<b>module</b>	Specified module address (1...255).	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
Conversion no.	–	8	7	6	5	4	3	2	1

## Notes

Indicating the ADC is made bitwise, so that the conversion of several converters can be started simultaneously. For instance, when starting the AD converters 1 and 3 the bit pattern 0101b (decimal notation 5) must be transferred.

You can start a conversion with the instruction **P2\_SYNCALL** synchronously with other measurements, if you have released the module with **SYNCENABLE** for synchronization.

Several conversions can also be executed synchronously, if you have released the corresponding modules with **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

[ADCF](#), [READADCF](#), [READ\\_ADCF4](#), [READ\\_ADCF8](#), [READ\\_ADCF4\\_PACKED](#), [READ\\_ADCF8\\_PACKED](#), [WAIT\\_EOCF](#)

## To be used for the modules

Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG 'Declaration

EVENT:
  START_CONV(1,1) 'Start AD conversion
  WAIT_EOCF(1,1) 'Wait for end of conversion
  value = READADCF(1,1) 'Read the value from the ADC
```

## START\_CONV

## SYNC\_MODE

**SYNC\_MODE** determines on the specified module the type of synchronization with other modules, especially for burst-measurement sequences.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

SYNC_MODE (module, mode)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>mode</code>	Synchronization mode of the module (0...3): 0: no synchronization (default setting). 1: Synchronization as master module. 2: Synchronization as slave module. 3: Synchronization by a signal at the external EVENT input of any module (except CPU module).	LONG

### Notes

As soon as synchronized modules (in the mode 2 or 3) receive a synchronizing signal, they start simultaneously a conversion on all channels (the same function as with **START\_CONV**). This conversion can be part of a single measurement or of a burst-measurement sequence.

#### Mode "Master / Slave"

Only one master module is allowed. As soon as the master module starts a conversion – either as a single conversion (**START\_CONV**) or as part of a burst-measurement sequence – it immediately sends a synchronizing signal.

If slave modules receive the signal of the master module, they start conversion simultaneously on all channels.

For synchronized burst-measurement sequences you first have to send the instruction **BURST\_START** to the slave modules and then to the master module. With each conversion the master module sends a synchronizing signal, so that all conversions of the measurement sequences are running simultaneously on all synchronized modules.

#### Mode "Event"

Even-synchronized modules start a conversion when a signal arrives at a (released) event input of any module. Even signals of non-synchronized modules are allowed; a signal at the event input of the CPU module is ignored.

An event input is released by the instruction **EVENTENABLE**.

For each measurement in a measurement sequence an event signal is necessary. If you have set a burst-measurement sequence of 10000 measurements, then 10000 events have to arrive so that the measurement sequence can be completely processed.

If you synchronize burst-measurement sequences on several modules, you should initialize the modules to the same number of measurements with **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

BURST\_INIT, BURST\_CSTART, BURST\_READ, BURST\_READ\_PACKED, BURST\_START, BURST\_STATUS, SET\_GAIN

## To be used for the modules

Aln-F-4/14 Rev. B, Aln-F-8/14 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc
#DEFINE length 10000
#DEFINE module 1
#DEFINE sampleperiod 40      '1      MHz      measurement      rate
[=1/(25ns*40)]

DIM i AS LONG
DIM DATA_1[length], DATA_2[length], DATA_3[length] AS LONG
DIM DATA_4[length], DATA_5[length], DATA_6[length] AS LONG
DIM DATA_7[length], DATA_8[length], DATA_9[length] AS LONG
DIM DATA_10[length], DATA_11[length], DATA_12[length] AS LONG

INIT:
  SYNC_MODE(module,1)      'Master module
  SYNC_MODE(module+1,2)    'Slave module
  SYNC_MODE(module+2,2)    'Slave module
  REM Prepare and start burst-measurements for 4 channels each
  BURST_INIT(module,3,sampleperiod,length)
  BURST_INIT(module+1,3,sampleperiod,length)
  BURST_INIT(module+2,3,sampleperiod,length)
  BURST_START(module+1)    'Start burst-measurement slave
  BURST_START(module+2)    'Start burst-measurement slave
  BURST_START(module)      'Start burst-measurement master
  GLOBALDELAY=800         'Find trigger point with 50 kHz

EVENT:
  PAR_1=BURST_STATUS(module) 'Amount of measurements still to be
                             'executed
  IF (PAR_1=0) THEN END     'Burst-measurement finish - then
execute                     'FINISH

FINISH:
  REM Copy the last data of all 4 channels
  BURST_READ(module,1,1,length,DATA_1,1)
  BURST_READ(module,2,1,length,DATA_2,1)
  BURST_READ(module,3,1,length,DATA_3,1)
  BURST_READ(module,4,1,length,DATA_4,1)
  BURST_READ(module+1,1,1,length,DATA_5,1)
  BURST_READ(module+1,2,1,length,DATA_6,1)
  BURST_READ(module+1,3,1,length,DATA_7,1)
  BURST_READ(module+1,4,1,length,DATA_8,1)
  BURST_READ(module+2,1,1,length,DATA_9,1)
  BURST_READ(module+2,2,1,length,DATA_10,1)
  BURST_READ(module+2,3,1,length,DATA_11,1)
  BURST_READ(module+2,4,1,length,DATA_12,1)
```

## WAIT\_EOC

**WAIT\_EOC** waits, until the last A/D conversion has finished.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
WAIT_EOC (module)
```

### Parameters

*module* Specified module address (1...255).

LONG

### See also

[ADC](#), [ADC16](#), [SET\\_MUX](#), [START\\_CONV](#), [READADC](#)

### To be used for the modules

(LP)SH-8(-FI), AIn-16/14-C Rev. A, AIn-32/12 Rev. B, AIn-32/14 Rev. A, AIn-32/16 Rev. B, AIn-32/16 Rev. C, AIn-8/12 Rev. A, AIn-8/12 Rev. B, AIn-8/14 Rev. A, AIn-8/16 Rev. A, AIn-8/16 Rev. B, AIn-8/16 Rev. C, AO-16/8-12

### Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM value1 AS LONG 'Declaration

INIT:
  SET_MUX(1,0) 'Set multiplexer to input 1
  REM Wait here for the settling of the multiplexer1
  REM For 12-bit ADCs: 3µs;
  REM For AIn-8/16 Rev. B, AIn-32/16 Rev. B: 14µs

EVENT:
  START_CONV(1) 'Start AD conversion
  WAIT_EOC(1) 'Wait for end of conversion
  value1 = READADC(1) 'Read value from the ADC
```

Another possibility to wait for the settling time is the loop described below; it can only be used in high-priority processes:

```
time = READ_TIMER()
DO
  UNTIL (READ_TIMER()-time>120) 'wait 25ns*120=3µs
```

For AIn-8/16 Rev. B, AIn-32/16 Rev.:

```
UNTIL (READ_TIMER()-time>560) 'Wait 25ns*560=14µs
```

---

1. The settling time can be bridged by e.g. a couple of *ADbasic* instructions, which do not run a measurement on that A/D module where the multiplexer has been set.

**WAIT\_EOCF** waits until the end of conversion on all specified F-ADCs.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
```

```
WAIT_EOCF (module, adc_no)
```

## Parameters

**module** Specified module address (1...255). LONG

**adc\_no** 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.

## See also

[ADCF](#), [START\\_CONV](#), [READADCF](#), [READ\\_ADCF4](#), [READ\\_ADCF8](#),  
[READ\\_ADCF4\\_PACKED](#), [READ\\_ADCF8\\_PACKED](#)

## To be used for the modules

Aln-F-4/12 Rev. A, Aln-F-4/14 Rev. B, Aln-F-4/16 Rev. A, Aln-F-8/12 Rev. A, Aln-F-8/14 Rev. B, Aln-F-8/16 Rev. A

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG           'Declaration

EVENT:
  START_CONV(1,1)           'Start AD conversion
  WAIT_EOCF(1,1)            'Wait for end of conversion
  value = READADCF(1,1)     'Read value from ADC
```

## WAIT\_EOCF

### 3.3 Pro I: Output Modules

The include file `<ADwinPRO_ALL.inc>` includes all functions and procedures, which are necessary to get access to the *ADwin-Pro* I D/A modules. If you include this file with the *ADbasic* instruction

```
#INCLUDE ADwinPRO_ALL.inc
```

in your *ADbasic* program you will be able to apply all functions and procedures.

On the following pages all `<ADwinPRO_ALL.inc>` functions will be described more detailed. In addition an easy example is presented for every function.

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

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



**DAC** outputs an (analog) voltage on the channel of the specified module, that corresponds to the indicated digital value.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
DAC(module, dac_no, value)
```

## Parameters

module	Specified module address (1...255).	LONG
dac_no	Number of the output.	LONG
value	value to output (0...65535).	LONG

This procedure is characterized by a sequence of several commands:

<b>WRITEDAC</b>	→	<b>START_DAC</b>
Transfer digital value into DAC register		Start D/A conversion

## See also

[START\\_DAC](#), [WRITEDAC](#)

## To be used for the modules

AOut-16/8-12, AOut-4/16 Rev. A, AOut-4/16 Rev. B, AOut-4/16 Rev. C,  
AOut-8/16 Rev. A, AOut-8/16 Rev. B, AOut-8/16 Rev. C

## Example

```
REM Digital proportionl controller
#include ADwinPRO_ALL.inc
#include ADwinPRO_ALL.inc
DIM sp, dev AS LONG      'Declaration
DIM g, actuate AS LONG   'Declaration

EVENT:
  sp = PAR_1              'setpoint
  g = PAR_2              'Gain
  dev = sp - ADC(1,1)     'Calculate control deviation
  actuate = dev * g       'Calculate actuating value
  DAC(1,1,actuate)        'Output of actuating value
```

## DAC

## FG\_CONTROL

**FG\_CONTROL** starts or stops the function generator (output of values) on the selected output channels of the module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
FG_CONTROL(module,output,run_mode)
```

## Parameters

module	Specified module address (1...255).	LONG
output	Bit pattern (0...1111b), which determines the output channels to be set, see table: Bit = 0: Do not change the operation mode. Bit = 1: Set the operation mode <code>run_mode</code> .	LONG
run_mode	Set operation mode (0...2): 0: Start output immediately (if function generator was stopped); Continue output and cancel soft stop (if function generator is running). 1: Stop output immediately (hard stop). 2: Stop output after the last buffer value (soft stop).	LONG

Bit No.	31...8	3	2	1	0
DAC No.	–	4	3	2	1

## Notes

This instruction can only be used when the function generator mode of the module is activated with **FG\_MODE**. The instruction affects all selected channels synchronously (see below).

After start a function generator runs – as long as its parameters keep unchanged – independently from the processor module. It will therefore not be affected by a processor boot; but a **FG\_MODE** in a newly started process will stop all running function generators.

If the function generator has output the last buffer value, it starts again with the first value of the buffer (without any time delay).

A "soft stop" stops the function generator as soon as the last buffer value is output. The modes 0 (start) and 1 (hard stop) cancel a previous soft stop immediately; in mode 1 the output stops at once, in mode 0 the output is continued (and not restarted with the first buffer value).

On a channel where the function generator is stopped or not active, a value may be output with **DAC**. A time delay may occur (see **FG\_MODE**). If the instruction **DAC** is given after the soft stop, it will be executed after the function generator has finally stopped.

After setting the operation mode 0 the output of values starts within 1 µs. You can query with **FG\_STATUS** whether the output has already started.

## See also

[DAC](#), [FG\\_DEF](#), [FG\\_DELAY](#), [FG\\_MODE](#), [FG\\_READ\\_INDEX](#), [FG\\_STATUS](#), [FG\\_WRITE](#)

## To be used for the modules

AOut-4/16 Rev. C

## Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM values[100] AS LONG      'Function array to be output
DIM i AS LONG                'loop variable

LOWINIT:
  FG_MODE(1, 1)              'enable function generator mode
  FG_DEF(1, 1, 200, 100)     'DacNo=1, startadr=200, memsize=100
  FOR i=1 TO 100
    values[i]=32768+i*327.67 'compute sawtooth function 0-10V
  NEXT i
  FG_WRITE(1, 200, 100, values, 1) 'Startadr=200, length=100,
                                   'array=values, array_idx=1
  FG_DELAY(1, 1, 80000)      'DacNo = 1, scale=80000 (= 1kHz output
                              'rate); with 100 values there is a
                              'sawtooth period of 100ms

INIT:
  FG_CONTROL(1,01b, 0)       'immediately start output at DAC 1

EVENT:
  par_1=FG_READ_INDEX(1, 1) 'put position pointer of function
                              'generator (DAC1) into PAR_1

FINISH:
  FG_CONTROL(1, 01111b,2)    'softstop for all channels =
                              'output all remaining values of the
                              'current period.
  DAC(1, 1, 49152)           'set DAC to 5V after finishing the
                              'function generator

  REM Wait until all function generators have stopped. This
  REM waiting loop ensures that all memory values are output,
  REM before the FG mode is disabled.
  DO
    UNTIL (FG_STATUS(1)=0)

  FG_MODE(1,0)               'disable function generator mode = set
                              'to normal mode
```

## FG\_DEF

For the output channel of a specified module, **FG\_DEF** defines the start address and the size of the internal buffer for the function generator mode.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

FG_DEF(module, DACno, startadr, memsize)
```

### Parameters

module	Specified module address (1...255).	LONG
DACno	Numbers of the output channel (1...4).	LONG
startadr	Start address of the buffer (1...0FFFFFFh).	LONG
memsize	Size of the buffer (1...0FFFFFFh) in 16-bit values.	LONG

### Notes

This instruction can only be used when the function generator mode of the module is activated with **FG\_MODE**.

For each channel a buffer must be set up. The function generator of a channel may only be started when its buffer is determined.

The internal buffer can receive up to 1048575 ( $2^{20}-1$ ) values à 16-bit each. The start address and the size of the buffer can individually be selected for each channel; there is no automatic check for range violations.

The definition of a buffer can be changed while the function generator is in progress. The change is effective (without any time delay), as soon as the function generator has output the last value of the current buffer range. The new buffer area must then contain valid data.

### See also

[FG\\_CONTROL](#), [FG\\_DELAY](#), [FG\\_MODE](#), [FG\\_READ\\_INDEX](#), [FG\\_STATUS](#), [FG\\_WRITE](#)

### To be used for the modules

AOut-4/16 Rev. C

### Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM values[100] AS LONG    'Function array to be output
DIM i AS LONG              'loop variable

LOWINIT:
  FG_MODE(1, 1)            'enable function generator mode
  FG_DEF(1, 1, 200, 100)   'DacNo=1, startadr=200, memsize=100
  FOR i=1 TO 100
    values[i]=32768+i*327.67 'compute sawtooth function 0-10V
  NEXT i
  FG_WRITE(1, 200, 100, values, 1) 'Startadr=200, length=100,
                                   'array=values, array_idx=1
  FG_DELAY(1, 1, 80000)    'DacNo = 1, scale=80000 (= 1kHz output
                           'rate); with 100 values there is a
                           'sawtooth period of 100ms
```



**FG\_DELAY** sets the output rate of function generator on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
FG_DELAY (module, DACno, scale)
```

## Parameters

module	Specified module address (1...255).	LONG
DACno	Numbers (1...4) of the output channel.	LONG
scale	Scale factor ( $80 \dots 2,68 \cdot 10^8 = 2^{29} - 1$ ) for setting the output rate using the formula: Output rate = 80MHz / scale.	LONG

## Notes

This instruction can only be used when the function generator mode of the module is activated with **FG\_MODE**.

The output rate can be set using the scale factor ranging from 0.15 Hz to 1.0 MHz with a resolution of 12.5 ns.

The instruction changes the output rate immediately.

Note that the output rate of the function generator is controlled by an individual timer of the AOut-M2 module and that the output rate is therefore not synchronous to the timer of the processor module.



## See also

[FG\\_CONTROL](#), [FG\\_DEF](#), [FG\\_MODE](#), [FG\\_READ\\_INDEX](#), [FG\\_STATUS](#), [FG\\_WRITE](#)

## To be used for the modules

AOut-4/16 Rev. C

## Beispiel

```
#INCLUDE ADwinPRO_ALL.inc

#define 1 1 'Address of Aout-module
DIM values[100] AS LONG 'Function array to be output
DIM i AS LONG 'loop variable

LOWINIT:
  FG_MODE(1, 1) 'enable function generator mode
  FG_DEF(1, 1, 200, 100) 'DacNo=1, startadr=200, memsize=100
  FOR i=1 TO 100
    values[i]=32768+i*327.67 'compute sawtooth function 0-10V
  NEXT i
  FG_WRITE(1, 200, 100, values, 1) 'Startadr=200, length=100,
    'array=values, array_idx=1
  FG_DELAY(1, 1, 80000) 'DacNo = 1, scale=80000 (= 1kHz output
    'rate); with 100 values there is a
    'sawtooth period of 100ms
```

## FG\_MODE

**FG\_MODE** enables or disables the function generator mode on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
```

```
FG_MODE(module,mode)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>mode</code>	Set the operation mode: 0: function generator OFF = default setting. 1: function generator ON.	LONG

### Notes

The function generator instructions (FG...) should always be used in the following order. If possible, use the noted program section:

- Enable function generator mode: **FG\_MODE** **LOWINIT:**
- Set parameters: **FG\_DEF**, **FG\_DELAY**, **LOWINIT:**  
**FG\_WRITE**
- Start function generator: **FG\_CONTROL** **INIT:**
- Query if needed: **FG\_READ\_INDEX**, **INIT:**  
**FG\_STATUS** **EVENT:**  
**FINISH:**
- Stop function generator: **FG\_CONTROL** **FINISH:**
- Disable function generator: **FG\_MODE** **FINISH:**

Disabling the function generator immediately stops all outputs of the active function generators. If, instead, a complete output of the buffer data is desired, querying the function generator status with the instructions **FG\_STATUS** must be made.

Please note, that with each enabling (even without previous disabling) the parameters have to be newly set by **FG\_DEF**, **FG\_DELAY** and **FG\_WRITE**. The initialization should be done from a single process only.

When the function generator mode is disabled, all channels can be accessed by using the instructions **DAC**, **WRITEDAC** and **START\_DAC**.

If the function generator mode is enabled only those channels may be accessed by the instructions **DAC**, **WRITEDAC** and **START\_DAC** where the function generator is stopped. Each of these instruction may then cause an occasional time delay (jitter) of up to 1µs.

### See also

[FG\\_CONTROL](#), [FG\\_DEF](#), [FG\\_DELAY](#), [FG\\_READ\\_INDEX](#), [FG\\_STATUS](#), [FG\\_WRITE](#)

### To be used for the modules

AOut-4/16 Rev. C



## Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM values[100] AS LONG      'Function array to be output
DIM i AS LONG                'loop variable

LOWINIT:
  FG_MODE(1, 1)              'enable function generator mode
  FG_DEF(1, 1, 200, 100)     'DacNo=1, startadr=200, memsize=100
  FOR i=1 TO 100
    values[i]=32768+i*327.67 'compute sawtooth function 0-10V
  NEXT i
  FG_WRITE(1, 200, 100, values, 1) 'Startadr=200, length=100,
                                   'array=values, array_idx=1
  FG_DELAY(1, 1, 80000)      'DacNo = 1, scale=80000 (= 1kHz output
                              'rate); with 100 values there is a
                              'sawtooth period of 100ms

INIT:
  FG_CONTROL(1,01b, 0)       'immediately start output at DAC 1

EVENT:
  par_1=FG_READ_INDEX(1, 1) 'put position pointer of function
                              'generator (DAC1) into PAR_1

FINISH:
  FG_CONTROL(1, 01111b,2)    'softstop for all channels =
                              'output all remaining values of the
                              'current period.
  DAC(1, 1, 49152)           'set DAC to 5V after finishing the
                              'function generator

  REM Wait until all function generators have stopped. This
  REM waiting loop ensures that all memory values are output,
  REM before the FG mode is disabled.
  DO
    UNTIL (FG_STATUS(1)=0)

  FG_MODE(1,0)               'disable function generator mode = set
                              'to normal mode
```

## FG\_READ\_INDEX

**FG\_READ\_INDEX** returns the position pointer of a specified function generator. The pointer is defined as the absolute memory address of the value which has been output at last.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = FG_READ_INDEX(module, DACno)
```

### Parameters

module	Specified module address (1...255).	LONG
DACno	Numbers (1...4) of the output channel.	LONG
ret_val	Memory address (1...0FFFFFFh) as pointer to the value, which has been output at last.	LONG

### Notes

The position pointer is valid only if the function generator mode of the module is activated with **FG\_MODE** and the function generator of this channel is running, too (status query see **FG\_STATUS**). After stopping the function generator the position pointer keeps the memory address of the last output value.

The output position in the buffer of a channel results from the difference of the return value `ret_val` and the start address `startadr` of the buffer indicated at **FG\_DEF**: `ret_val - startadr`.

### See also

[FG\\_CONTROL](#), [FG\\_DEF](#), [FG\\_DELAY](#), [FG\\_MODE](#), [FG\\_STATUS](#), [FG\\_WRITE](#)

### To be used for the modules

AOut-4/16 Rev. C

### Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM values[100] AS LONG 'Function array to be output
DIM i AS LONG           'loop variable

LOWINIT:
  FG_MODE(1, 1)           'enable function generator mode
  FG_DEF(1, 1, 200, 100)  'DacNo=1, startadr=200, memsize=100
  FOR i=1 TO 100
    values[i]=32768+i*327.67 'compute sawtooth function 0-10V
  NEXT i
  FG_WRITE(1, 200, 100, values, 1) 'Startadr=200, length=100,
                                   'array=values, array_idx=1
  FG_DELAY(1, 1, 80000)    'DacNo = 1, scale=80000 (= 1kHz output
                           'rate); with 100 values there is a
                           'sawtooth period of 100ms

INIT:
  FG_CONTROL(1, 01b, 0)    'immediately start output at DAC 1

EVENT:
  par_1=FG_READ_INDEX(1, 1) 'put position pointer of function
                             'generator (DAC1) into PAR_1
```

**FG\_STATUS** returns the status of all function generators of the module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = FG_STATUS(module)
```

## Parameters

<b>module</b>	Specified module address (1...255).	LONG
<b>ret_val</b>	Bit pattern which indicates the status of the function generators. A bit is allocated to each output channel. Bit=0: function generator not active. Bit=1: function generator is active.	LONG

Bit No.	31...8	3	2	1	0
DAC No.	–	4	3	2	1

## Notes

This instruction can only be used when the function generator mode of the module is activated with **FG\_MODE**.

If a function generator is stopped with the instruction **FG\_CONTROL**(..., 2) (soft stop), its bit in **ret\_val** will only be reset to 0 (zero) when the last value in the range is output.

## See also

[FG\\_CONTROL](#), [FG\\_DEF](#), [FG\\_DELAY](#), [FG\\_MODE](#), [FG\\_READ\\_INDEX](#), [FG\\_WRITE](#)

## To be used for the modules

AOut-4/16 Rev. C

## Example

```
#INCLUDE ADwinPRO_ALL.inc

#define 1 1          'Address of Aout-module

LOWINIT:
    FG_MODE(1, 1)    'enable function generator mode
    FG_DEF(1, 1, 200, 100) 'DacNo=1, startadr=200, memsize=100
    ...

INIT:
    FG_CONTROL(1, 01b, 0) 'immediately start output at DAC 1

EVENT:
    ...

FINISH:
    ...
    DO
        UNTIL (FG_STATUS(1)=0) 'wait until all channels are stopped.

        FG_MODE(1, 0)          'disable function generator mode = set
                                'to normal mode
```

## FG\_STATUS

## FG\_WRITE

**FG\_WRITE** transfers an even number of data of an array to a specified address in the buffer of the module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

FG_MODE(module, startadr, length, array[], array_idx)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>startadr</code>	Start address (0...0FFFFFFh) in the buffer. Beginning here, data are stored.	LONG
<code>length</code>	Amount (2...0FFFFFFh) of array elements to be transferred. The amount must be even numbered.	LONG
<code>array[]</code>	Source array whose values are transferred to the memory.	ARRAY LONG
<code>array_idx</code>	Index of the first source array element to be transferred.	LONG

### Notes

This instruction can only be used when the function generator mode of the module is activated with **FG\_MODE**.

The parameters `startadr` and `length` must correspond to the settings made under **FG\_DEF**.

The source array must have `length+array_idx` elements at least. From the elements of the source array the lower word (bits 0...15) are transferred to the buffer. The bits 16...31 are not considered.

The number of transferred field elements must be even; an odd number of elements could get the *ADwin* system into an instable operation mode.

In a high-priority process you are only allowed to transfer as much data with **FG\_WRITE** at the same time to the module, so that the workload of the *ADwin* system is not higher than 100%. If the workload is exceeded, the communication to the PC will become unstable (time-out). You will not find this restriction in the section **LOWINIT**: and in low-priority processes.

### See also

[FG\\_CONTROL](#), [FG\\_DEF](#), [FG\\_DELAY](#), [FG\\_MODE](#), [FG\\_READ\\_INDEX](#), [FG\\_STATUS](#)

### To be used for the modules

AOut-4/16 Rev. C



## Example

```
#INCLUDE ADwinPRO_ALL.inc

#DEFINE 1 1          'Address of Aout-module
DIM values[100] AS LONG 'Function array to be output
DIM i AS LONG        'loop variable

LOWINIT:
  FG_MODE(1, 1)      'enable function generator mode
  FG_DEF(1, 1, 200, 100) 'DacNo=1, startadr=200, memsize=100
  FOR i=1 TO 100
    values[i]=32768+i*327.67 'compute sawtooth function 0-10V
  NEXT i
  FG_WRITE(1, 200, 100, values, 1) 'Startadr=200, length=100,
    'array=values, array_idx=1
  FG_DELAY(1, 1, 80000) 'DacNo = 1, scale=80000 (= 1kHz output
    'rate); with 100 values there is a
    'sawtooth period of 100ms

INIT:
  FG_CONTROL(1,01b, 0) 'immediately start output at DAC 1

EVENT:
  ...
```

## START\_DAC

**START\_DAC** starts the conversion or output of all DACs on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
START_DAC (module)
```

### Parameters

module      Specified module address (1...255).

LONG

### See also

WRITEDAC, DAC

### To be used for the modules

AOut-16/8-12, AOut-4/16 Rev. A, AOut-4/16 Rev. B, AOut-4/16 Rev. C,  
AOut-8/16 Rev. A, AOut-8/16 Rev. B, AOut-8/16 Rev. C

### 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                    'Declaration  
INIT:  
    i=0  
  
EVENT:  
    WRITEDAC (1,1,i)              'Set output register DAC1  
    WRITEDAC (1,2,65535-i)       'Set output register DAC2  
    START_DAC (1)                'Start output on all DACs  
    INC (i)  
    IF (i=65535) THEN i=0
```



**WRITEDAC** writes a digital value into the output register of a DAC on the specified module. The conversion into output voltage is started by the instruction **START\_DAC**.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

WRITEDAC (module, dac_no, value)
```

## Parameters

module	Specified module address (1...255).	LONG
dac_no	Number of the output.	LONG
value	Value to output (0...4095 with 12-bit DAC, 0...65535 with 16-bit DAC).	LONG

## See also

[START\\_DAC](#), [DAC](#)

## To be used for the modules

AOut-16/8-12, AOut-4/16 Rev. A, AOut-4/16 Rev. B, AOut-4/16 Rev. C, AOut-8/16 Rev. A, AOut-8/16 Rev. B, AOut-8/16 Rev. C

## 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 4 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], DATA_3[1000] AS LONG
DIM DATA_4[1000] AS LONG

INIT:
    i=1

EVENT:
    WRITEDAC(1,1,DATA_1[i]) 'Set output register DAC1
    WRITEDAC(1,2,DATA_2[i]) 'Set output register DAC2
    WRITEDAC(1,3,DATA_3[i]) 'Set output register DAC3
    WRITEDAC(1,4,DATA_4[i]) 'Set output register DAC4
    START_DAC(1) 'Start output on all DACs
    INC(i)
    IF (i>1000) THEN i=1
```

## WRITEDAC

### 3.4 Pro I: Digital Modules

The include file `<ADWPDIO.INC>` includes all functions and procedures, which are necessary to get access to the *ADwin-Pro* digital I/O-modules. If you include this file with the *ADbasic* instruction

```
#INCLUDE ADwinPRO_ALL.inc
```

in your *ADbasic* program you can use all functions and procedures of this file. On the following pages all functions of the file `ADWPDIO.INC` will be described more detailed. In addition an easy application example illustrates each function.

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

It is presumed that application examples use the module address 1 for digital modules.



**CNT\_CLEAR** sets the counter values of one or more counters on the specified module to the value 0 (zero).

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
```

```
CNT_CLEAR(module,pattern)
```

## Parameters

**module** Specified module address (1...255). LONG

**pattern** Bit pattern, assignment to the counters, see table. LONG  
 Bit = 0: No function.  
 Bit = 1: Reset counter.

Module	Bit no.					
	15 ... 4	3	2	1	0	
Pro-CNT-16/16	–	4, 8, 12, 16	3, 7, 11, 15	2, 6, 10, 14	1, 5, 9, 13	
Pro-CNT-8/32	–	4, 8	3, 7	2, 6	1, 5	
Pro-CNT-16/32	16 ... 5	4	3	2	1	
Pro-CNT-VR4						
Pro-CNT-VR2PW2						
Pro-CNT-PW4	–	4	3	2	1	
Pro-CO4-T						
Pro-CO4-I						
Pro-CO4-D						

## To be used for the modules

CNT-16/16(-I), CNT-16/32(-I), CNT-8/32(-I), CNT-PW4(-I), CNT-VR2PW2, CNT-VR4(-I), CNT-VR4L(-I), CO4

## Example! Only to be used for the module Pro-CNT-VR4

```
#INCLUDE ADwinPRO_ALL.inc
```

### INIT:

```
CNT_SETMODE(1,01111b)  'Set counters 1...4 to
                        'clock/direction evaluation
CNT_CLEAR(1,01111b)    'Set values of counters 1...4 to 0
CNT_ENABLE(1,01111b)   'Enable counters 1...4
```

## CNT\_CLEAR

**CNT\_ENABLE**

**CNT\_ENABLE** enables or disables one or more counters on the specified module.

**Syntax**

```
#INCLUDE ADwinPRO_ALL.inc  
CNT_ENABLE(module,pattern)
```

**Parameters**

**module** Specified module address (1...255). LONG

**pattern** Enable counters according to the bit pattern, for assignment of the counters see table. LONG

Bit = 0: Disable counter / block.  
Bit = 1: Enable counter / release.

Module	Bit no.						
	15	...	4	3	2	1	0
Pro-CNT-16/16		–		4, 8, 12, 16	3, 7, 11, 15	2, 6, 10, 14	1, 5, 9, 13
Pro-CNT-8/32		–		4, 8	3, 7	2, 6	1, 5
Pro-CNT-16/32	16	...	5	4	3	2	1
Pro-CNT-VR4							
Pro-CNT-VR2PW2							
Pro-CNT-PW4		–		4	3	2	1
Pro-CO4-T							
Pro-CO4-I							
Pro-CO4-D							

**To be used for the modules**

CNT-16/16(-I), CNT-16/32(-I), CNT-8/32(-I), CNT-PW4(-I), CNT-VR2PW2, CNT-VR4(-I), CNT-VR4L(-I), CO4

**Example**

Only to be used for the module Pro-CNT-VR4!

```
#INCLUDE ADwinPRO_ALL.inc
```

**INIT:**

```
CNT_SETMODE(1,01000b) 'Set counter 4 to clock/direction  
                        'evaluation, all other counters to  
                        '4 edge evaluation  
CNT_CLEAR(1,01000b)   'Set counter values of counter 4 to 0  
CNT_ENABLE(1,01000b)  'Enable counter 4, disable all others
```



**CNT\_LATCH** transfers the current counter values of one or more counters on the specified module into the respective latch register(s) (= to latch).

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
CNT_LATCH(module,pattern)
```

## Parameters

**module** Specified module address (1...255). LONG

**pattern** Latch counter values according to bit pattern, for assignment of the counters, see table. LONG

Bit = 0: No function.  
Bit = 1: Transfer counter values into latch register .

Module	Bit no.						
	15	...	4	3	2	1	0
Pro-CNT-16/16		–		4, 8, 12, 16	3, 7, 11, 15	2, 6, 10, 14	1, 5, 9, 13
Pro-CNT-8/32		–		4, 8	3, 7	2, 6	1, 5
Pro-CNT-16/32	16	...	5	4	3	2	1
Pro-CNT-VR4							
Pro-CNT-VR2PW2							
Pro-CNT-PW4		–		4	3	2	1
Pro-CO4-T							
Pro-CO4-I							
Pro-CO4-D							

## See also

[CNT\\_READLATCH16](#), [CNT\\_READLATCH32](#)

## To be used for the modules

CNT-16/16(-I), CNT-16/32(-I), CNT-8/32(-I), CNT-PW4(-I), CNT-VR2PW2, CNT-VR4(-I), CNT-VR4L(-I), CO4

## Example

Only to be used for the module Pro-CNT-VR4

```
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG

INIT:
value = ADC(1,1)           'Get measurement value
IF (value>49151) THEN
CNT_LATCH(1,00011b)       'Latch counters 1 and 2
ENDIF
REM Calculate difference
PAR_1 = CNT_READLATCH32(1,00001b) - CNT_READLATCH32(1,00010b)
```



## CNT\_READ16

**CNT\_READ16** returns the current counter value of a 16-bit counter on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = CNT_READ16(module, cnt_no)
```

### Parameter

module	Specified module address (1...255).	LONG
cnt_no	Counter number.	LONG
ret_val	Current counter value (16-bit value).	LONG

### Example

The function is characterized by a sequence of two instructions, which are illustrated below.

CNT_LATCH	→	CNT_READLATCH16
Copy current count value to the latch register		Read out latch register

For special applications you can also use these instructions instead of **CNT\_READ16**.

### See also

[CNT\\_READ32](#), [CNT\\_LATCH](#), [CNT\\_READLATCH16](#)

### To be used for the modules

CNT-16/16(-I)

### Example

```
#INCLUDE ADwinPRO_ALL.inc
EVENT:
PAR_1 = CNT_READ16(1,1) 'Get current value of counter 1
PAR_2 = CNT_READ16(1,2) 'Get current value of counter 2
```

**CNT\_READ32** returns the current counter value of a 32-bit counter on the specified module.

## Syntax

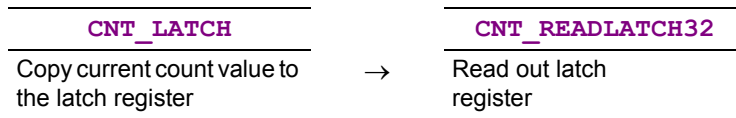
```
#INCLUDE ADwinPRO_ALL.inc
ret_val = CNT_READ32(module, cnt_no)
```

## Parameters

module	Specified module address (1...255).	LONG
cnt_no	Counter number.	LONG
ret_val	Current counter value (32-bit value).	LONG

## Notes

This function is characterized by a sequence of two instructions, which are illustrated below.



For special applications you can also use these instructions instead of **CNT\_READ32**.

## See also

[CNT\\_READ16](#), [CNT\\_LATCH](#), [CNT\\_READLATCH32](#)

## To be used for the modules

CNT-8/32(-I), CNT-PW4(-I), CNT-VR2PW2, CNT-VR4(-I), CNT-VR4L(-I)

## Example

```
#INCLUDE ADwinPRO_ALL.inc
EVENT:
PAR_1 = CNT_READ32(1, 3)  'Get current value of counter 3
PAR_2 = CNT_READ32(1, 4)  'Get current value of counter 4
```

## CNT\_READ32

## CNT\_READLATCH16

**CNT\_READLATCH16** returns the value from the latch register of a 16-bit counter on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
ret_val = CNT_READLATCH16(module, cnt_no)
```

### Parameters

module	Specified module address (1...255).	LONG
cnt_no	Counter number.	LONG
ret_val	Current counter value (16-bit value).	LONG

### Notes

In order to get the current counter value, it has to be copied into the latch register before with **CNT\_LATCH** and then can be read out.

### See also

[CNT\\_LATCH](#), [CNT\\_READ16](#)

### To be used for the modules

CNT-16/16(-I)

### Example

```
#INCLUDE ADwinPRO_ALL.inc  
DIM value AS LONG  
  
INIT:  
value = ADC(1,1)           'Get measurement value  
IF (value>49151) THEN CNT_LATCH(1,3)  
                           'Latch counters 1 and 2  
PAR_1 = CNT_READLATCH16(1,1) - CNT_READLATCH16(1,2)  
                           'Difference counters 1 and 2
```



**CNT\_READLATCH32** returns the value from the latch register of a 32-bit counter on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = CNT_READLATCH32 (module, cnt_no)
```

## Parameters

module	Specified module address (1...255).	LONG
cnt_no	Counter number.	LONG
ret_val	Current counter value (32-bit value).	LONG

## Notes

In order to get the current counter value, it has to be copied into the latch register before with **CNT\_LATCH** and then can be read out.

## See also

[CNT\\_LATCH](#), [CNT\\_READ32](#)

## To be used for the modules

CNT-8/32(-I), CNT-PW4(-I), CNT-VR2PW2, CNT-VR4(-I), CNT-VR4L(-I)

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG

INIT:
value = ADC(1,1)           'Get measurement value
IF (value>49151) THEN CNT_LATCH(1,3)
                           'Latch counters 1 and 2
PAR_1 = CNT_READLATCH32(1,1) - CNT_READLATCH32(1,2)
                           'Difference counters 1 and 2
```

## CNT\_READLATCH32

## CNT\_SETMODE

**CNT\_SETMODE** sets the operating mode of all counters on the specified module, four edge evaluation or clock and direction input.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
CNT_SETMODE(module,pattern)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>pattern</code>	Bit pattern for setting the operating mode of the counters: . Bit = 0: Mode four edge evaluation. Bit = 1: Mode clock and direction input.	LONG

Module	Bit 3	Bit 2	Bit 1	Bit 0
Pro-CNT-VR4	4	3	2	1
Pro-CNT-VR2PW2				

### Notes

The up/down counters are operated in one of two modes. The four edge evaluation mode is used for quadrature encoders whose signals are shifted by 90 degrees. The mode for clock and direction input is used for all other encoders.

### To be used for the modules

CNT-VR2PW2, CNT-VR4(-I), CNT-VR4L(-I)

### Example

```
#INCLUDE ADwinPRO_ALL.inc  
  
INIT:  
  CNT_SETMODE(1,1100b)      'Counter 3 and 4 to clock/direction  
                             'evaluation, all others to  
                             'four edge evaluation  
  CNT_CLEAR(1,1100b)        'Set values of counters 3 and 4 to 0  
  CNT_ENABLE(1,1100b)       'Enable counters 3 and 4,  
                             'disable all others
```

**CO4\_CLEARENABLE** enables the external input CLR of one or more counters.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
CO4_CLEARENABLE(module,pattern)
```

## Parameters

<b>module</b>	Specified module address (1...255).	LONG
<b>pattern</b>	Bit pattern for counter assignment (see table). Bit = 0: No function. Bit = 1: Enable input CLR.	LONG

	Bit no.							
	7	6	5	4	3	2	1	0
Counter no.	4*	3*	2*	1*	4	3	2	1

\*Only in the operation mode "four edge evaluation":

Bit = 0: Clear counter, if the signals A, B, and CLR are set to "High"

Bit = 1: Clear counter, if CLR is set to "High".

## Note:

The external input may be enabled either with the instruction **CO4\_CLEARENABLE** or with **CO4\_LATCHENABLE**, but not with both instructions simultaneously!

## See also

[CO4\\_LATCHENABLE](#)

## To be used for the modules

CO4

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
CO4_SETMODE(1,1,1)      'Counter 1: CLK/DIR mode
CO4_LATCHENABLE(1,0)    'Disable latch-input for all counters
CO4_CLEARENABLE(1,1)    'Enable clear-input for counter 1
CNT_CLEAR(1,1)          'Clear counter 1
CNT_ENABLE(1,1)         'Start counter 1

EVENT:
PAR_1 = CO4_READ(1,1)   'Read out counter 1
```

## CO4\_CLEAREN- ABLE

## CO4\_GETSTATUS

**CO4\_GETSTATUS** returns the status of the input signals of a counter on the specified module as bit pattern.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
ret_val = CO4_GETSTATUS(module, cnt_no)
```

## Parameters

module	Specified module address (1...255).	LONG
cnt_no	Counter number.	LONG
ret_val	Bit pattern showing the status of the counter inputs (bits 31...7 have no meaning):	LONG

Bit no.						
6	5	4	3	2	1	0
Current status of a signal input			Status message (Bit = 1)			
(Pro-CO4-D: signal after the level converter)			Signal: Readable once		Error: Readable repeatedly	
Input B	Input A	Input CLR/LATCH	Signal LATCH exists	Signal CLR exists	Correlation error	Cable error

**Error (repeatedly readable status message):** Use the instruction **CO4\_RESETSTATUS** to delete the message.

Bit 0 = 1: Cable error; the differential signals (A & /A or B & /B or C & /C; C = CLR-/LATCH) have one of the following errors:

A cable is not connected (cable break), short circuit, signal voltage is too low, common-mode voltage is too high or slew rate is too low (< 0.33V/μs).

Bit 1 = 1: Correlation error; simultaneous modification of the signals A and B instead of a phase -shift.

**Signal (once readable status message):** The status message is cleared by reading out.

Bit 2 = 1: CLR signal exists (when it has been released by the instruction **CO4\_CLEARENABLE**).

Bit 3 = 1: LATCH signal exists (when it has been released by the instruction **CO4\_LATCHENABLE**).

## Notes

A cable error (bit 0) can only be detected at differential inputs! At TTL inputs these bits are always 0 (zero) and the instruction **CO4\_RESETSTATUS** has no effect.

Even if errors occur, the counters will not be stopped.

## See also

[CO4\\_RESETSTATUS](#)

## To be used for the modules

CO4

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM error AS LONG

INIT:
  CO4_SETMODE(1,1,0)      'Counter 1 four edge evaluation mode
  CNT_CLEAR(1,1)          'Clear counter 1
  CNT_ENABLE(1,1)         'Start counter 1
  CO4_RESETSTATUS(1,1)    'Clear status register
  error = 0               'Reset error code

EVENT:
  PAR_1 = CO4_READ(1,1)    'Read out counter 1
  PAR_2 = CO4_GETSTATUS(1,1) 'Read out counter 1
  IF (PAR_2 AND 1 = 1) THEN 'Cable error?
    INC PAR_3              'Amount of cable errors until now
    error = 1              'Set error code
  ENDIF
  IF (PAR_2 AND 2 = 2) THEN 'Correlation error?
    INC PAR_4              'Amount of correlation errors
    error = 1              'Set error code
  ENDIF
  PAR_5 = SHIFT_RIGHT(PAR_2 AND 16,4) 'Current status CLR input
  PAR_6 = SHIFT_RIGHT(PAR_2 AND 32,5) 'Current status input A
  PAR_7 = SHIFT_RIGHT(PAR_2 AND 64,6) 'Current status input B
  IF (error = 1) THEN      'Is error code set?
    CO4_RESETSTATUS(1,1)  'Reset status register
    error = 0              'Reset error code
  ENDIF
```

## CO4\_LATCHENABLE

**CO4\_LATCHENABLE** enables the external input LATCH of one or more counters on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
CO4_LATCHENABLE(module,pattern)
```

### Parameters

<b>module</b>	Specified module address (1...255).	LONG
<b>pattern</b>	Bit pattern for counter assignment (see table):	LONG
	Bit = 0: No function.	
	Bit = 1: Release input LATCH.	

Bit no.	Bit 3	Bit 2	Bit 1	Bit 0
Counter no.	4	3	2	1

### Notes

The external input may be enabled either with the instruction **CO4\_CLEARENABLE** or with **CO4\_LATCHENABLE**, but not with both instructions simultaneously!

### See also

[CO4\\_CLEARENABLE](#)

### To be used for the modules

CO4

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
  CO4_SETMODE(1,1,1)      'Counter 1: CLK/DIR mode
  CO4_CLEARENABLE(1,0)    'Disable clear-input for all counters
  CO4_LATCHENABLE(1,1)    'Release latch-input (counter 1)
  CNT_CLEAR(1,1)          'Clear counter 1
  CNT_ENABLE(1,1)         'Start counter 1

EVENT:
  PAR_1 = CO4_READLATCH(1,1) 'Read out counter 1
```

**CO4\_READ** returns the current counter value from the specified module.

## Syntax

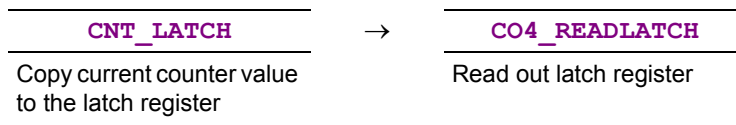
```
#INCLUDE ADwinPRO_ALL.inc
ret_val = CO4_READ(module, cnt_no)
```

## Parameters

module	Specified module address (1...255).	LONG
cnt_no	Counter number.	LONG
ret_val	Current counter value of the specified counter.	LONG

## Notes

This function is characterized by a sequence of two instructions, which are illustrated below.



For specific applications you can also use these instructions instead of **CO4\_READ**.

## See also

[CNT\\_LATCH](#), [CO4\\_READLATCH](#)

## To be used for the modules

CNT-16/32(-I), CO4

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
CO4_SETMODE(1,1,1)      'Counter 1: CLK/DIR mode
CNT_CLEAR(1,1)          'Clear counter 1
CNT_ENABLE(1,1)         'Start counter 1

EVENT:
PAR_1 = CO4_READ(1,1)   'Get current value of counter 1
```

## CO4\_READ

## CO4\_READ- LATCH

**CO4\_READLATCH** returns the value of the latch register of a counter on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = CO4_READLATCH(module, cnt_no)
```

### Parameters

module	Specified module address (1...255).	LONG
cnt_no	Counter number.	LONG
ret_val	Value from the latch register of the counter.	LONG

### Notes

In order to get the current counter value, it has to be latched before into the latch register with the instruction **CNT\_LATCH** and can then be read by **CO4\_READLATCH**.

### See also

[CNT\\_LATCH](#), [CO4\\_READ](#)

### To be used for the modules

CNT-16/32(-I), CO4

### Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM old, new AS LONG      'Dimensioning the variables

INIT:
old = 0                   'Initialize the variable
CO4_SETMODE(1,1,1)        'Counter 1: CLK/DIR mode
CNT_CLEAR(1,1)            'Reset counter 1
CNT_ENABLE(1,1)           'Start counter 1

EVENT:
CNT_LATCH(1,1)            'Latch counter 1 and..
new = CO4_READLATCH(1,1)  'read out latch
PAR_1 = new - old         'Calculate difference
(f = impulses / time)
old = new                 'Save new counter value as old ones
```



**CO4\_RESETSTATUS** clears the status register of one or more counters on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
CO4_RESETSTATUS (module, pattern)
```

## Parameters

<b>module</b>	Specified module address (1...255).	LONG
<b>pattern</b>	Bit pattern for counter assignment (see table). Bit = 0: No function. Bit = 1: Clear status register bits 0 and 1.	LONG

Bit no.	Bit 3	Bit 2	Bit 1	Bit 0
Counter no.	4	3	2	1

## Notes

The status register contains the status of a counter's input signals (read out using **CO4\_GETSTATUS**).

## See also

[CO4\\_GETSTATUS](#)

## To be used for the modules

CO4

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM error AS LONG

INIT:
CO4_SETMODE (1,1,0)           'Counter 1:four edge mode
CNT_CLEAR (1,1)               'Clear counter 1
CNT_ENABLE (1,1)              'Start counter 1
CO4_RESETSTATUS (1,1)         'Clear status register
error = 0                     'Reset error code

EVENT:
PAR_1 = CO4_READ (1,1)        'Read out counter 1
PAR_2 = CO4_GETSTATUS (1,1)    'Get input status of counter 1
IF (PAR_2 AND 1 = 1) THEN 'Cable error?
    INC PAR_3                  'Amount of cable errors until now
    error = 1                  'Set error code
ENDIF
IF (PAR_2 AND 2 = 2) THEN 'Correlation errors?
    INC PAR_4                  'Set number of correlation errors
    error = 1                  'Set error code
ENDIF
PAR_5 = SHIFT_RIGHT (PAR_2 AND 16,4) 'Current status CLR input
PAR_6 = SHIFT_RIGHT (PAR_2 AND 32,5) 'Current status input A
PAR_7 = SHIFT_RIGHT (PAR_2 AND 64,6) 'Current status input B
IF (error = 1) THEN           'Is error code set?
    CO4_RESETSTATUS (1,1)     'Reset status register
    error = 0                 'Reset error code
ENDIF
```

## CO4\_RESETSTATUS

## CO4\_SET\_LATCHMODE

**CO4\_SET\_LATCHMODE** determines the mode of the latch-inputs for all counters on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

CO4_SET_LATCHMODE (module, pattern)
```

### Parameters

**module** Specified module address (1...255). LONG

**pattern** Bit pattern for the latch-mode of the counters (see LONG tables); the default setting is 00000000b.

	Destination register for software latch (CNT_LATCH)				Copy latch content into additional latch			
Bit no. in <i>pattern</i>	7	6	5	4	3	2	1	0
Counter no.	4	3	2	1	4	3	2	1

Bit value	Destination register for software latch (CNT_LATCH)	Copy latch content into additional latch
Bit = 0	Counter value is transferred into latch for negative edges: Latches 5...8	With each positive edge the latch content for negative edges (5...8) is copied into the additional latch 9...12.
Bit = 1	Counter values are transferred into latches for positive edges: Latches 1...4	With each negative edge the latch contents for positive edges (1...4) is copied into the additional latch 9...12.

### Notes

This instruction is only useful in connection with PWM analysis.

You should have experience with PWM analysis on an *ADwin-Pro* system, before you use this instruction. If you have further questions, call our support.

The instruction **CO4\_SET\_LATCHMODE** determines

- if the contents of the latch for positive edges is saved in the additional latch, or the contents of the latch for negative edges. This makes it possible to acquire a single fast change from wide to small pulse width.
- into which destination register the counter values are transferred at a software latch.

### See also

[CNT\\_LATCH](#)

### To be used for the modules

CO4



**CO4\_SETMODE** sets the count mode of a counter on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
CO4_SETMODE(module, cnt_no, mode)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>cnt_no</code>	Counter number (1...4).	LONG
<code>mode</code>	Counter mode: 0: Four edge evaluation (A and B signal inputs). 1: Clock and direction (CLK and DIR inputs). 2: PWM analysis.	LONG

## Notes

The counters can be operated in 3 different modes. The 4 edge evaluation is used for quadrature encoders with two signals phase-shifted by 90°, whereas the clock and direction inputs are used for general applications. In the PWM mode the analysis of a PWM signal is possible, that means frequency, period width and impulse duration as well as pause duration (duty cycle) can be determined.

## See also

[CNT\\_CLEAR](#), [CNT\\_ENABLE](#)

## To be used for the modules

CO4

## CO4\_SETMODE

## Example

```
#INCLUDE ADwinPRO_ALL.inc
#DEFINE refCLK 40E6
DIM rise, rise_old, fall, fall_old, T AS LONG

INIT:
    rise_old = 0
    fall_old = 0
    CO4_SETMODE(1,1,2)      'Counter 1: PWM analysis
    CNT_CLEAR(1,1)          'Clear counter 1
    CNT_ENABLE(1,1)         'Start counter 1

EVENT:
    rise = CO4_READLATCH(1,1) 'Read out latch 1
                                '(pos. edge, counter 1)
    fall = CO4_READLATCH(1,5) 'Read out latch 5
                                '(neg. edge, counter 1)
    IF (rise <> rise_old) THEN 'Pos. edge detected?
        IF (fall <> fall_old) THEN 'Neg. edge detected,
                                'that means is PWM signal low?
            PAR_1 = fall - rise    'Pulse duration in periods of the
                                'reference clock
            PAR_2 = rise - fall_old 'Pause duration in periods of the
                                'reference clock
        ELSE
            'No neg. edge detected, that means
            'PWM signal = HIGH?
            PAR_1 = fall - rise_old 'Pulse duration in periods of the
                                'reference clock
            PAR_2 = rise - fall    'Pause duration in periods of the
                                'reference clock
        ENDIF
    ENDIF
    T = PAR_1 + PAR_2            'Period duration in periods of the
                                'reference clock
    FPAR_1 = refCLK / T          'Frequency of the PWM signal
    FPAR_2 = PAR_1 * 100 / T    'Duty cycle in percent
    rise_old = rise              'Save latch
    fall_old = fall              'Save latch
```

**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
DIG_LATCH (module)
```

## Parameters

**module** Specified module address (1...255). LONG

## Notes

It is recommended for the modules Pro-DIO-32 and Pro-DIO-32 Rev. B to first program the channels using the instructions **DIGPROG1** and **DIGPROG2** as inputs or outputs.

Depending which module you use, the instruction transfers the following information:

Module	Input signal to input latches	Output latches to outputs
Pro-OPT-16	x	–
Pro-REL-16 Pro-TRA-16	–	x
Pro-DIO-32 Pro-DIO-32 Rev. B	x	x

If the module is released for synchronization by **SYNCENABLE**, the instruction **P2\_SYNCALL** has the same functions as the instruction **DIG\_LATCH**.

## See also

[DIG\\_READLATCH1](#), [DIG\\_READLATCH2](#), [DIG\\_WRITELATCH1](#), [DIG\\_WRITELATCH2](#), [DIG\\_WRITELATCH32](#), [EXTLCH\\_ENABLE](#)  
[DIGPROG1](#), [DIGPROG2](#), [DIGIN\\_WORD1](#), [DIGIN\\_WORD2](#), [DIGOUT](#), [DIGOUT\\_WORD1](#), [DIGOUT\\_WORD2](#)  
[DIGIN\\_LONG\\_F](#), [DIGOUT\\_BITS\\_F](#), [DIGOUT\\_F](#), [DIGOUT\\_LONG\\_F](#)  
[GET\\_DIGOUT\\_LONG](#),  
[GET\\_DIGOUT\\_WORD1](#), [GET\\_DIGOUT\\_WORD2](#)  
[SYNCALL](#), [SYNCENABLE](#)

## To be used for the modules

DIO-32, DIO-32 Rev. B, OPT-16 Rev. A, OPT-16 Rev. B, REL-16 Rev. A, REL-16 Rev. B, TRA-16 Rev. A, TRA-16 Rev. B

## DIG\_LATCH

### Example

```
#INCLUDE ADwinPRO_ALL.inc
```

#### INIT:

```
DIGPROG1(1,0FFFFh)      'DIO15:00 (low-word) of DIO-32-  
                          'module as output  
DIGPROG2(1,0)            'DIO31:16 (high-word) of DIO-32-  
                          'module as input  
DIG_WRITELATCH1(1,0)     'Set all output bits to 0
```

#### EVENT:

```
DIG_LATCH(1)             'Latch inputs, output contents of the  
                          'output latch  
...                       'more program steps  
PAR_1 = DIG_READLATCH2(1) 'Read in high-word and output at ...  
DIG_WRITELATCH1(1,PAR_1) 'the next event in the low-word.
```

**DIG\_READLATCH1** returns the lower 16 bits (bit 0...bit 15) from the latch register for the digital inputs of the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
DIG_READLATCH1 (module)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
---------------------	-------------------------------------	------

## Notes

It is recommended to first program the specified channels as inputs using the instruction **DIGPROG1**.

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

- DIGIN\_WORD1
- DIGIN\_WORD2
- P2\_SYNCALL (when activated)

## See also

[DIG\\_LATCH](#), [DIG\\_READLATCH2](#), [DIG\\_WRITELATCH1](#), [DIG\\_WRITELATCH2](#), [DIG\\_WRITELATCH32](#), [EXTLCH\\_ENABLE](#)  
[DIGPROG1](#), [DIGPROG2](#), [DIGIN\\_WORD1](#), [DIGIN\\_WORD2](#), [DIGIN\\_LONG\\_F](#)  
[SYNCALL](#), [SYNCENABLE](#)

## To be used for the modules

DIO-32, DIO-32 Rev. B, OPT-16 Rev. A, OPT-16 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADWINPRO.INC
DIM value AS LONG

INIT:
    REM Set DIO15:00 of modules 1+2 as inputs
    DIGPROG1 (1,0)
    DIGPROG1 (2,0)
    SYNCENABLE (1,dio,1)      'Enable synchronization of module 1
    SYNCENABLE (2,dio,1)      'Enable synchronization of module 2

EVENT:
    P2_SYNCALL ()             'Transfer the logic level at the
                              'digital inputs of both modules
                              'synchronously to the latch register
    PAR_1 = DIG_READLATCH1 (1) 'Read out temp. register of module 1
                              '(bits 0...15)
    PAR_2 = DIG_READLATCH1 (2) 'Read out temp. register of module 2
                              '(bits 0...15)
```

## DIG\_READLATCH1

## DIG\_READLATCH2

**DIG\_READLATCH2** returns the upper 16 bits (bit 16... bit 31) from the latch register for the digital inputs of the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
DIG_READLATCH2(module)
```

### Parameters

<code>module</code>	Specified module address (1...255).	<div style="border: 1px solid black; padding: 2px; display: inline-block;">LONG</div>
---------------------	-------------------------------------	---

### Notes

It is recommended to first program the specified channels as inputs using the instruction **DIGPROG2**.

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

- DIGIN\_WORD1
- DIGIN\_WORD2
- P2\_SYNCALL (when activated)

### See also

[DIG\\_LATCH](#), [DIG\\_READLATCH1](#), [DIG\\_WRITELATCH1](#), [DIG\\_WRITELATCH2](#), [DIG\\_WRITELATCH32](#), [EXTLCH\\_ENABLE](#)  
[DIGPROG1](#), [DIGPROG2](#), [DIGIN\\_WORD1](#), [DIGIN\\_WORD2](#), [DIGIN\\_LONG\\_F](#)  
[SYNCALL](#), [SYNCENABLE](#)

### To be used for the modules

DIO-32, DIO-32 Rev. B

### Example

```
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADWINPRO.INC
DIM value AS LONG

INIT:
    REM Set DIO31:16 of modules 1+2 as inputs
    DIGPROG2(1,0)
    DIGPROG2(2,0)
    SYNCENABLE(1,dio,1)      'Enable synchronization of module 1
    SYNCENABLE(2,dio,1)      'Enable synchronization of module 2

EVENT:
    P2_SYNCALL()             'Transfer the logic level at the
                              'digital inputs of both modules
                              'synchronously to the latch register
    PAR_1 = DIG_READLATCH2(1) 'Read out temp. register of module 1
                              '(bits 16...31)
    PAR_2 = DIG_READLATCH2(2) 'Read out temp. register of module 2
                              '(bits 16...31)
```



**DIG\_Writelatch1** writes a value into the lower 16 bits (bit 0...15) of the latch register for the digital outputs of the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

DIG_Writelatch1(module,pattern)
```

## Parameters

**module** Specified module address (1...255). LONG

**pattern** Bit pattern. Each bit corresponds to a digital output (see table). LONG

Bit no.	31:16	15	14	13	...	2	1	0
Output no.	–	15	14	13	...	2	1	0

## Notes

For the modules Pro-DIO-32 and Pro-DIO-32 Rev. B, the specified channels must be first programmed as outputs using the instruction **DIGPROG1**.

You can set the value of the latch register for the digital outputs with the following instructions:

- DIGOUT
- DIGOUT\_WORD1
- DIGOUT\_WORD2
- DIG\_Writelatch1
- DIG\_Writelatch2
- DIG\_Writelatch32

The instruction must not be used in combination with **DIG\_Writelatch2**. If you want to change bits both in the low and in the high word of the latch register, please use the instruction **DIG\_Writelatch32**.

## See also

[DIG\\_Latch](#), [DIG\\_ReadLatch1](#), [DIG\\_Writelatch2](#), [DIG\\_Writelatch32](#), [EXTLCH\\_ENABLE](#)

[DIGPROG1](#), [DIGPROG2](#), [DIGOUT](#), [DIGOUT\\_WORD1](#), [DIGOUT\\_WORD2](#), [GET\\_DIGOUT\\_WORD1](#), [GET\\_DIGOUT\\_WORD2](#)

## To be used for the modules

DIO-32, DIO-32 Rev. B, REL-16 Rev. A, REL-16 Rev. B, TRA-16 Rev. A, TRA-16 Rev. B

## DIG\_Writelatch1



### Example

```
#INCLUDE ADWINPRO.INC
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG

INIT:
  REM DIO-32 only: Set DIO15:00 of the module as outputs
  DIGPROG1(1,0FFFFh)
  DIGPROG1(2,0FFFFh)

  SYNCENABLE(1,dio,1)      'Enable synchronization of digital
                             'module 1
  SYNCENABLE(2,dio,1)      '... and digital module 2
  DIG_WRITELATCH1(1,1)     'Set lowest bit in the output
                             'latch register
  DIG_WRITELATCH1(2,1)     'Set lowest bit in the output
                             'latch register

EVENT:
  value = ADC(1,1)          'Measurement data acquisition
  IF (value > 3000) THEN    'Limit value exceeded?
    P2_SYNCALL()           'Output values from the latch
                             'registers
  ENDIF
```

**DIG\_Writelatch2** writes a value into the upper 16 bits (bit 16...31) of the latch register for the digital outputs of the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
DIG_Writelatch2 (module, pattern)
```

## Parameters

**module** Specified module address (1...255). LONG

**pattern** Bit pattern. Each bit corresponds to a digital output LONG (see table).

Bit no.	31:16	15	14	13	...	2	1	0
Output no.	–	31	30	29	...	18	17	16

## Notes

The specified channels must be first programmed as outputs using the instruction **DIGPROG2**.

You can set the value of the latch register for the digital outputs with the following instructions:

- DIGOUT
- DIGOUT\_WORD1
- DIGOUT\_WORD2
- DIG\_Writelatch1
- DIG\_Writelatch2
- DIG\_Writelatch32

You must not use this instruction in combination with **DIG\_Writelatch1**. If you want to change bits both in the low and in the high word of the latch register, please use the instruction **DIG\_Writelatch32**.

## See also

[DIG\\_Latch](#), [DIG\\_ReadLatch1](#), [DIG\\_Writelatch1](#), [DIG\\_Writelatch32](#), [EXTLCH\\_Enable](#)

[DIGPROG1](#), [DIGPROG2](#), [DIGOUT](#), [DIGOUT\\_WORD1](#), [DIGOUT\\_WORD2](#), [GET\\_DIGOUT\\_WORD1](#), [GET\\_DIGOUT\\_WORD2](#)

## To be used for the modules

DIO-32, DIO-32 Rev. B

## DIG\_Writelatch2



### Example

```
#INCLUDE ADWINPRO.INC
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG

INIT:
  REM DIO-32 only: Set DIO31:16 of the module as outputs
  DIGPROG2 (1,0FFFFh)
  DIGPROG2 (2,0FFFFh)

  SYNCENABLE (1,dio,1)      'Enable synchronization of module 1
  SYNCENABLE (2,dio,1)      '... and module 2
  DIG_Writelatch2 (1,1)      'Set bit 16 in the output latch
                              'register
  DIG_Writelatch2 (2,10000b) 'Set bit 20 in the output latch
                              'register

EVENT:
  value = ADC (1,1)          'Measurement data acquisition
  IF (value > 3000) THEN      'Limit value exceeded?
    P2_SYNCALL ()            'Output values from the latch
                              'registers
  ENDIF
```

**DIG\_Writelatch32** writes a 32-bit value into the long-word (bits 31...0) of the latch on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
DIG_Writelatch32 (module, pattern)
```

## Parameters

**module** Specified module address (1...255). LONG

**pattern** Bit pattern. Each bit corresponds to a digital output (see table). LONG

Bit no.	31	30	29	...	2	1	0
Output no.	31	30	29	...	18	17	16

## Notes

The specified channels must be first programmed as outputs using the instructions **DIGPROG1** and **DIGPROG2**.

You can set the value of the latch register for the digital outputs with the following instructions:

- DIGOUT
- DIGOUT\_WORD1
- DIGOUT\_WORD2
- DIG\_Writelatch1
- DIG\_Writelatch2
- DIG\_Writelatch32

## See also

[DIG\\_LATCH](#), [DIG\\_READLATCH1](#), [DIG\\_Writelatch1](#), [DIG\\_Writelatch2](#), [EXTLCH\\_ENABLE](#)

[DIGOUT](#), [DIGOUT\\_WORD1](#), [DIGOUT\\_WORD2](#), [DIGPROG1](#), [DIGPROG2](#), [GET\\_DIGOUT\\_WORD1](#), [GET\\_DIGOUT\\_WORD2](#)

## To be used for the modules

DIO-32, DIO-32 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
DIGPROG1 (1, 0FFFFh)      'DIO15:00 (low-word) of DIO-32-
                           'module as output
DIGPROG2 (1, 0FFFFh)      'DIO31:16 (high-word) of DIO-32-
                           'module as output

EVENT:
DIG_LATCH (1)              'Output information of the output
                           latch
                           'on a DIO-32 board
DIG_Writelatch32 (1, PAR_1) 'Write long-word into output latch
```

## DIG\_Writelatch32

## DIGIN\_LONG\_F

**DIGIN\_LONG\_F** returns the status of the inputs (bits 31...00) of the specified module as bit pattern.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = DIGIN_LONG_F(module)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>ret_val</code>	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 the instructions **DIGPROG1** and **DIGPROG2**.

### See also

[DIG\\_LATCH](#), [DIGIN\\_WORD1](#), [DIGIN\\_WORD2](#), [DIGPROG1](#),  
[DIGPROG2](#)  
[DIGOUT\\_LONG\\_F](#), [DIGOUT\\_WORD1](#), [DIGOUT\\_WORD2](#)

### To be used for the modules

DIO-32 Rev. B

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
DIGPROG1(1,0)          'Set DIO15:00 (Low-Word) as inputs
DIGPROG2(1,0)          'Set DIO31:16 (High-Word) as inputs

EVENT:
PAR_1 = DIGIN_LONG_F(1) 'Read all inputs
```

**DIGIN\_WORD1** returns the status of the inputs 0...15 of the specified module as bit pattern.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = DIGIN_WORD1(module)
```

## Parameters

<b>module</b>	Specified module address (1...255).	LONG
<b>ret_val</b>	Bit pattern. Each of the bits 15...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.	31:16	15	14	...	2	1	0
Input	–	15	14	...	2	1	0

## Notes

It is recommended for the modules Pro-DIO-32 and Pro-DIO-32 Rev. B to first program the channels as inputs using the instructions **DIGPROG1** and **DIGPROG2**.

## See also

[DIGIN\\_WORD2](#), [DIGOUT](#), [DIGOUT\\_WORD1](#), [DIGOUT\\_WORD2](#), [DIGPROG1](#), [DIGPROG2](#)

## To be used for the modules

DIO-32, DIO-32 Rev. B, OPT-16 Rev. A, OPT-16 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADwinPRO_ALL.inc
DIM DATA_1[10000] AS LONG AS FIFO

INIT:
  REM DIO32 only: Set DIO15:00 (Low-Word) as inputs
  DIGPROG1(4,0)

EVENT:
  IF (DIGIN_WORD1(4) AND 14 = 14) THEN
    'Query, if inputs 1, 2 and 3
    'on module 4 are set
    DATA_1 = ADC(1,1)      'Measurement data acquisition
  ENDIF
```

The bit pattern of the decimal value 14 (which is ...01110b) enables you to recognize which digital inputs are set, here the inputs 1, 2 and 3.

In *ADbasic* you can also enter numbers in binary notation. Please add the letter "b" to the bit pattern. The line in the example above would then be as follows:

```
IF (DIGIN_WORD1(4) AND 1110b = 1110b) THEN
```

## DIGIN\_WORD1

## DIGIN\_WORD2

**DIGIN\_WORD2** returns the status of the inputs 16...31 of the specified module as bit pattern.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
ret_val = DIGIN_WORD2(module)
```

## Parameters

module	Specified module address (1...255).	LONG
ret_val	Bit pattern. Each of the bits 15...0 corresponds to the input status of a digital input (see table). Bit = 0: Low level. Bit = 1: High level.	LONG

Bit no.	31:16	15	14	...	2	1	0
Input	–	31	30	...	18	17	16

## Notes

It is recommended to first program the specified channels as inputs using the instruction **DIGPROG2**.

## See also

[DIGIN\\_WORD1](#), [DIGOUT](#), [DIGOUT\\_WORD1](#), [DIGOUT\\_WORD2](#), [DIGPROG1](#), [DIGPROG2](#)

## To be used for the modules

DIO-32, DIO-32 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc  
#INCLUDE ADwinPRO_ALL.inc  
DIM DATA_1[10000] AS LONG AS FIFO  
  
INIT:  
    DIGPROG2(4,0)                'DIO31:16 (high word) as inputs  
  
EVENT:  
    'Query if inputs 16, 18 and 21 are set on module 4  
    IF (DIGIN_WORD2(4) AND 39 = 39) THEN  
        DATA_1 = ADC(1,1)        'Measurement data acquisition  
    ENDIF
```

The bit pattern of the decimal value 39 (which is ...0100101b) enables you to recognize which digital inputs are set, here the inputs 16, 18 and 21.

In *ADbasic* you can also enter numbers in binary notation. Please add the letter "b" to the bit pattern. The line in the example above would then be as follows:

```
IF (DIGIN_WORD2(4) AND 0100101b = 0100101b) THEN
```



**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
DIGOUT (module, output, value)
```

## Parameters

module	Specified module address (1...255).	LONG
output	Number of the output to be set (0...31).	LONG
value	New status of the selected output: 0: Low level. 1: High level.	LONG

## Notes

The instruction can also be used for the module Pro-DIO-32 Rev. B. But we recommend to use the instruction **DIGOUT\_F**, because it works faster and needs essentially less program memory.

The specified channel must be first programmed as output using the instruction **DIGPROG1** or **DIGPROG2**.

This procedure is characterized by a sequence of two instructions, which are illustrated below.

<b>GET_DIGOUT_</b> <b>WORD1/2</b>	→	...	→	<b>DIGOUT_</b> <b>WORD1/2</b>
Read current status of all digital outputs		Set or reset speci- fied bit position		Write back manipulated value

In two parallel processes which have different priority you are not allowed to apply this function with the same module:

A process with low priority can be interrupted in the middle of a **DIGOUT** sequence by a process with higher priority. If the process with higher priority changes the setting of the digital outputs during this interruption, these changes will be lost when the low priority Process writes back the manipulated values with **DIGOUT\_WORD**.

Several parallel processes with high priority can apply the function **DIGOUT** without any problems, because processes with high priority do not interrupt each other.

## See also

[DIGOUT\\_F](#), [DIGOUT\\_BITS\\_F](#), [DIGOUT\\_WORD1](#), [DIGOUT\\_WORD2](#), [DIGPROG1](#), [DIGPROG2](#), [GET\\_DIGOUT\\_WORD1](#), [GET\\_DIGOUT\\_WORD2](#)

## To be used for the modules

DIO-32, DIO-32 Rev. B, REL-16 Rev. A, REL-16 Rev. B, TRA-16 Rev. A, TRA-16 Rev. B

## DIGOUT



## Example

```
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG

INIT:
    REM DIO32 only: Set channels as outputs
    DIGPROG1(1,0FFFFh)      'DIO15:00 (low word) as outputs
    DIGPROG2(1,0FFFFh)      'DIO31:16 (high word) as outputs

EVENT:
    value = ADC(1,1)         'Measurement data acquisition
    IF (value < 100) THEN    'If value is below limit
        DIGOUT(1,2,0)       'reset digital output 2
    ENDIF
```

**DIGOUT\_BITS\_F** sets the specified outputs of the specified module to the levels "high" or "low".

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
DIGOUT_BITS_F(module, set, clear)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>set</code>	Bit pattern that sets specified digital outputs to the level "high". Bit = 0: Output remains unchanged. Bit = 1: Set output to level "high".	LONG
<code>clear</code>	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 the instructions **DIGPROG1** and **DIGPROG2**.

You can set or clear any required outputs without changing the status of the remaining outputs. The logical combination of the corresponding registers is made with this instruction on hardware level. Thus it runs much faster than the instruction **DIGOUT**, that works on software level.

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.

## See also

**DIGOUT**, **DIGOUT\_F**, **DIGOUT\_WORD1**, **DIGOUT\_WORD2**, **DIGPROG1**, **DIGPROG2**

## To be used for the modules

DIO-32 Rev. B, TRA-16 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
  REM DIO32 only: Set channels as outputs
  DIGPROG1(1,0FFFFh)      'DIO15:00 (low word) als Ausgang
  DIGPROG2(1,0FFFFh)      'DIO31:16 (high word) als Ausgang

EVENT:
  IF (PAR_1 = 1) THEN      'Get condition
    DIGOUT_BITS_F(1,8888h,7777h) 'Set MSB of every byte,
                                'Clear all other bits
  ELSE
    DIGOUT_BITS_F(1,5555h,0AAAAh) 'Set odd-numbered bits,
                                'clear even-numbered bits
  ENDIF
```

## DIGOUT\_BITS\_F

## DIGOUT\_F

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

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
DIGOUT_F(module,output,value)
```

### Parameters

module	Specified module address (1...255).	LONG
output	Number of the output to be set (0...31).	LONG
value	New status of the selected output: 0: Low level. 1: High level.	LONG

### Notes

The specified channels must be first programmed as outputs using the instructions **DIGPROG1** and **DIGPROG2**.

The logical combination of the corresponding registers is made with this instruction on hardware level. Thus it runs much faster than the instruction **DIGOUT**, that works on software level.

### See also

[DIGOUT](#), [DIGOUT\\_BITS\\_F](#), [DIGOUT\\_WORD1](#), [DIGOUT\\_WORD2](#), [DIGPROG1](#), [DIGPROG2](#)

### To be used for the modules

DIO-32 Rev. B, TRA-16 Rev. B

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
    REM DIO32 only: Set channels as outputs
    DIGPROG1(1,0FFFFh)      'DIO15:00 (low word) as outputs
    DIGPROG2(1,0FFFFh)      'DIO31:16 (high word) as outputs

EVENT:
    IF (DIGIN_WORD1(1) AND 8000h = 1) THEN
        'Read low-word (bits 0...15) and
        'check, if MSB is set
        DIGOUT_F(2,31,0)    'If MSB is set, clear bit 31
    ELSE
        DIGOUT_F(2,31,1)    'If MSB is cleared, set bit 31
    ENDIF
```

With the 32-bit value the instruction **DIGOUT\_LONG\_F** sets or clears all outputs on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
DIGOUT_LONG_F(module,pattern)
```

## Parameters

<b>module</b>	Specified module address (1...255).	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 the instructions **DIGPROG1** and **DIGPROG2**.

## See also

[DIGOUT](#), [DIGOUT\\_F](#), [DIGOUT\\_BITS\\_F](#), [DIGOUT\\_WORD1](#),  
[DIGOUT\\_WORD2](#), [DIGPROG1](#), [DIGPROG2](#)

## To be used for the modules

DIO-32 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
DIGPROG1(1,0FFFFh)      'DIO15:00 (low word) as outputs
DIGPROG2(1,0FFFFh)      'DIO31:16 (high word) as outputs

EVENT:
DIGOUT_LONG_F(1,1000000) 'Output the value 1 million as binary
                        'value on DIOs
```

## DIGOUT\_LONG\_F

## DIGOUT\_WORD1

**DIGOUT\_WORD1** sets the digital outputs 0...15 on the specified module simultaneously to the specified levels.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
DIGOUT_WORD1(module, pattern)
```

### Parameters

<b>module</b>	Specified module address (1...255).	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...16	15	...	1	0
Output	—	15	...	1	0

### Notes

For the modules Pro-DIO-32 and Pro-DIO-32 Rev. B, the specified channels must be first programmed as outputs using the instruction **DIGPROG1**.

### See also

[DIG\\_LATCH](#), [DIG\\_READLATCH1](#), [DIG\\_READLATCH2](#), [DIG\\_WRITELATCH1](#), [DIG\\_WRITELATCH2](#), [DIG\\_WRITELATCH32](#), [EXTLCH\\_ENABLE](#)  
[DIGPROG1](#), [DIGPROG2](#), [DIGIN\\_WORD1](#), [DIGIN\\_WORD2](#), [DIGOUT](#), [DIGOUT\\_WORD2](#)  
[DIGIN\\_LONG\\_F](#), [DIGOUT\\_BITS\\_F](#), [DIGOUT\\_F](#), [DIGOUT\\_LONG\\_F](#)  
[GET\\_DIGOUT\\_LONG](#),  
[GET\\_DIGOUT\\_WORD1](#), [GET\\_DIGOUT\\_WORD2](#)

### See also

[DIGIN\\_WORD1](#)

### To be used for the modules

DIO-32, DIO-32 Rev. B, REL-16 Rev. A, REL-16 Rev. B, TRA-16 Rev. A, TRA-16 Rev. B

### Example

```
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG

INIT:
  REM DIO32 only: Set channels as outputs
  DIGPROG1(4,0FFFFh)      'DIO15:00 (low word) as outputs

EVENT:
  value = ADC(1,1)          'Measurement data acquisition
  IF (value > 3000) THEN    'Is limit value exceeded?
    DIGOUT_WORD1(4,111b)   'Set outputs 0, 1 and 2 of the module 4
                           'all other outputs are reset.
  ENDIF
```

**DIGOUT\_WORD2** sets the digital outputs 16...31 on the specified module simultaneously to the specified levels.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
DIGOUT_WORD2 (module, pattern)
```

## Parameters

<b>module</b>	Specified module address (1...255).	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...16	15	...	1	0
Output	–	31	...	17	16

## Notes

The specified channels must be first programmed as outputs using the instruction **DIGPROG2**.

## See also

DIG\_LATCH, DIG\_READLATCH1, DIG\_READLATCH2, DIG\_WRITELATCH1, DIG\_WRITELATCH2, DIG\_WRITELATCH32, EXTLCH\_ENABLE  
DIGPROG1, DIGPROG2, DIGIN\_WORD1, DIGIN\_WORD2, DIGOUT, DIGOUT\_WORD1  
DIGIN\_LONG\_F, DIGOUT\_BITS\_F, DIGOUT\_F, DIGOUT\_LONG\_F  
GET\_DIGOUT\_LONG,  
GET\_DIGOUT\_WORD1, GET\_DIGOUT\_WORD2

## See also

DIGIN\_WORD2

## To be used for the modules

DIO-32, DIO-32 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG

INIT:
    DIGPROG2 (1, 0FFFFh)      'DIO31:16 (high word) as outputs

EVENT:
    value = ADC (1, 1)        'Measurement data acquisition
    IF (value > 3000) THEN    'Is limit value exceeded?
        DIGOUT_WORD2 (4, 1011b) 'set outputs 16, 17 a. 19, all
                                'other outputs are reset!
    ENDIF
```

## DIGOUT\_WORD2

## DIGPROG1

**DIGPROG1** programs the digital channels 0...15 of the specified module as input or output.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
```

```
DIGPROG1 (module, pattern)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>pattern</code>	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.	Module	31...16	15	...	8	7	...	0
channel no.	DIO-32 Rev. A	–	15	...	08	07	...	00
	DIO-32 Rev. B	–	–	–	15:08	–	–	07:00

### Notes

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

Consider the different ways of programming the modules:

- Pro-DIO-32 Rev. A: Each of the channels can be individually set as input or output.
- Pro-DIO-32 Rev. B: The channels can only be set as inputs or outputs in groups of 8 (2 relevant bits only, the other bits are ignored).

### See also

[DIG\\_LATCH](#), [DIG\\_READLATCH1](#), [DIG\\_READLATCH2](#), [DIG\\_WRITELATCH1](#), [DIG\\_WRITELATCH2](#), [DIG\\_WRITELATCH32](#), [EXTLCH\\_ENABLE](#)

[DIGPROG2](#), [DIGIN\\_WORD1](#), [DIGIN\\_WORD2](#), [DIGOUT](#), [DIGOUT\\_WORD1](#), [DIGOUT\\_WORD2](#)

[DIGIN\\_LONG\\_F](#), [DIGOUT\\_BITS\\_F](#), [DIGOUT\\_F](#), [DIGOUT\\_LONG\\_F](#), [GET\\_DIGOUT\\_LONG](#), [GET\\_DIGOUT\\_WORD1](#), [GET\\_DIGOUT\\_WORD2](#)

### To be used for the modules

DIO-32, DIO-32 Rev. B

### Example

```
#INCLUDE ADwinPRO_ALL.inc
```

```
INIT:
```

```
DIGPROG1 (1, 11111111b) 'Configures channels 0...7 of DIO  
                        'module no. 1 as outputs and channels  
                        '8...15 as inputs
```



**DIGPROG2** programs all channels 16...31 of the specified module as inputs or outputs.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
DIGPROG2 (module, pattern)
```

## Parameters

<b>module</b>	Specified module address (1...255).	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.	Module	31...16	15	...	8	7	...	0
Channel no.	DIO-32 Rev. A	–	31	...	24	23	...	16
	DIO-32 Rev. B	–	–	–	31:24	–	–	23:16

## Notes

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

Consider the different ways of programming the modules:

- Pro-DIO-32 Rev. A: Each of the channels can be individually set as input or output.
- Pro-DIO-32 Rev. B: The channels can only be set as inputs or outputs in groups of 8 (2 relevant bits only, the other bits are ignored).

## See also

DIG\_LATCH, DIG\_READLATCH1, DIG\_READLATCH2, DIG\_WRITELATCH1, DIG\_WRITELATCH2, DIG\_WRITELATCH32, EXTLCH\_ENABLE

DIGPROG1, DIGIN\_WORD1, DIGIN\_WORD2, DIGOUT, DIGOUT\_WORD1, DIGOUT\_WORD2

DIGIN\_LONG\_F, DIGOUT\_BITS\_F, DIGOUT\_F, DIGOUT\_LONG\_F  
GET\_DIGOUT\_LONG,  
GET\_DIGOUT\_WORD1, GET\_DIGOUT\_WORD2

## To be used for the modules

DIO-32, DIO-32 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
    DIGPROG2 (1, 11111111b)  'Configures channels 16...23 of the
                              'DIO module no. 1 as outputs and
                              ' channels 24...31 as inputs
```

## DIGPROG2

## EXTLCH\_ENABLE

**EXTLCH\_ENABLE** enables or disables all latch-inputs on the specified module. The latch-inputs are selected by the corresponding counter number.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
EXTLCH_ENABLE(module, cnt_select)
```

### Parameters

<b>module</b>	Specified module address (1...255).	LONG
<b>cnt_select</b>	Bit pattern for selecting the counters and setting the latch status: Bit = 0: Disable latch-input. Bit = 1: Enable latch-input.	LONG

Bit no.	31:5	3	2	1	0
Counter no.	–	4	3	2	1

### Notes

In the folder <C:\ADwin\ADbasic\samples\_ADwin\_PRO> there is an example program <Pro-CNT-VR4-L-I\_CLK-DIR.BAS>.

### See also

[CNT\\_CLEAR](#), [CNT\\_ENABLE](#), [CNT\\_LATCH](#), [CNT\\_READ32](#), [CNT\\_READLATCH32](#), [CNT\\_SETMODE](#)

### To be used for the modules

CNT-VR4L(-I)

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
    EXTLCH_ENABLE(1, 0011b) 'Enable latch-inputs of counters 1+2,
                             'disable latch-inputs of counters 3+4
```

**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 = GET_DIGOUT_LONG(module)
```

## Parameters

module	Specified module address (1...255).	LONG
ret_val	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.

## See also

DIG\_LATCH, DIG\_READLATCH1, DIG\_READLATCH2, DIG\_WRITELATCH1, DIG\_WRITELATCH2, DIG\_WRITELATCH32, EXTLCH\_ENABLE  
DIGPROG1, DIGPROG2, DIGIN\_WORD1, DIGIN\_WORD2, DIGOUT, DIGOUT\_WORD1, DIGOUT\_WORD2  
DIGIN\_LONG\_F, DIGOUT\_BITS\_F, DIGOUT\_F, DIGOUT\_LONG\_F  
GET\_DIGOUT\_WORD1, GET\_DIGOUT\_WORD2

## To be used for the modules

DIO-32

## Example

```
#INCLUDE ADwinPRO_ALL.inc
```

### EVENT:

```
PAR_1 = GET_DIGOUT_LONG(1) 'return bits 31:00 from the latch
```

## GET\_DIGOUT\_LONG

## GET\_DIGOUT\_WORD1

**GET\_DIGOUT\_WORD1** returns the lower word (bits 0...15) of the output-latch (register for digital outputs) on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
ret_val = GET_DIGOUT_WORD1(module)
```

### Parameters

module	Specified module address (1...255).	LONG
ret_val	Contents of the output-latch (bits 15:00).	LONG

### Notes

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

### See also

[DIG\\_LATCH](#), [DIG\\_READLATCH1](#), [DIG\\_READLATCH2](#), [DIG\\_WRITELATCH1](#), [DIG\\_WRITELATCH2](#), [DIG\\_WRITELATCH32](#), [EXTLCH\\_ENABLE](#)  
[DIGPROG1](#), [DIGPROG2](#), [DIGIN\\_WORD1](#), [DIGIN\\_WORD2](#), [DIGOUT](#), [DIGOUT\\_WORD1](#), [DIGOUT\\_WORD2](#)  
[DIGIN\\_LONG\\_F](#), [DIGOUT\\_BITS\\_F](#), [DIGOUT\\_F](#), [DIGOUT\\_LONG\\_F](#)  
[GET\\_DIGOUT\\_LONG](#), [GET\\_DIGOUT\\_WORD2](#)

### To be used for the modules

DIO-32, DIO-32 Rev. B, REL-16 Rev. A, REL-16 Rev. B, TRA-16 Rev. A, TRA-16 Rev. B

### Example

```
#INCLUDE ADwinPRO_ALL.inc  
DIM value AS LONG  
  
INIT:  
value = GET_DIGOUT_WORD1(1) 'Read current value of the output  
                             'register  
value = value AND 0FFFFh 'Set LSB (bit 0) to 0  
DIGOUT_WORD1(value)       'Return changed value
```

**GET\_DIGOUT\_WORD2** returns the upper word (bits 16...31) of the output-latch (register for digital outputs) of the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = GET_DIGOUT_WORD2 (module)
```

## Parameters

module	Specified module address (1...255).	LONG
ret_val	Contents of the output-latch (bits 31:16).	LONG

## Notes

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

## See also

DIG\_LATCH, DIG\_READLATCH1, DIG\_READLATCH2, DIG\_WRITELATCH1, DIG\_WRITELATCH2, DIG\_WRITELATCH32, EXTLCH\_ENABLE  
DIGPROG1, DIGPROG2, DIGIN\_WORD1, DIGIN\_WORD2, DIGOUT, DIGOUT\_WORD1, DIGOUT\_WORD2  
DIGIN\_LONG\_F, DIGOUT\_BITS\_F, DIGOUT\_F, DIGOUT\_LONG\_F  
GET\_DIGOUT\_LONG, GET\_DIGOUT\_WORD1

## To be used for the modules

DIO-32, DIO-32 Rev. B

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM value AS LONG

INIT:
value = GET_DIGOUT_WORD2 (1) 'Read current value of the output
                             'register
value = value AND 0FFFDh 'Set bit 17 to 0
DIGOUT_WORD2 (value)      'Return changed value
```

## GET\_DIGOUT\_WORD2

## PWM\_ENABLE

**PWM\_ENABLE** enables or disables all internal counters of the specified module. The counters are selected by the corresponding PWM output number.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

PWM_ENABLE(module,output)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>output</code>	Bit pattern for selecting the PWM outputs and for setting the counter status: Bit = 0: Disable counter. Bit = 1: Enable counter.	LONG

Bit no.	31:5	3	2	1	0
PWM channel	–	4	3	2	1

### Notes



This instruction *does not* change the level of the PWM outputs. If you want to change the level of the PWM outputs, use the instruction **PWM\_OUT** and afterwards stop the corresponding internal counters with **PWM\_ENABLE**.

As soon as and as long as the counters are stopped, the PWM outputs cannot be changed.

### See also

[PWM\\_OUT](#), [PWM\\_SET](#)

### To be used for the modules

PWM-4(-I)

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
    PWM_ENABLE(1,1)           'Enable PWM output 1 for outputting
    PWM_SET(1,1,0,2500,2500)  'Set PWM signal for output 1
```

**PWM\_OUT** sets a specified PWM output channel on the specified module to the level "high" or "low".

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

PWM_OUT(module, channel, level)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>channel</code>	Number of the PWM output channel (1...4).	LONG
<code>level</code>	Level of the channel to be set: 0: Set to level Low. 1: Set to level High.	LONG

## Notes

This instruction has a function only when the corresponding counter of the specified channel is enabled.

## See also

[PWM\\_ENABLE](#), [PWM\\_SET](#)

## To be used for the modules

PWM-4(-I)

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
    PWM_ENABLE(1,3)           'Enable counters of the PWM outputs
                              '1 and 2
    PWM_SET(1,1,0,2500,2500)  'Set PWM signal for output 1
    PWM_OUT(1,2,1)           'Set PWM output 2 to the locical value
                              '"1".
```

## PWM\_OUT

## PWM\_SET

**PWM\_SET** makes the settings for a specified PWM output channel on the specified module.

These are:

- Factor of the prescaler.
- Value of the low-time.
- Value of the high-time.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
PWM_SET(module, channel, prescale, low, high)
```

### Parameters

module	Specified module address (1...255).	LONG
channel	Number of the PWM output channel (1...4).	LONG
prescale	Exponent (0...7) for the factor of the prescaler, see equation.	LONG
low	Value of the low-time (1...32768), see equation.	LONG
high	Value of the high-time (1...32768), see equation.	LONG

### Notes

The output frequency (after prescaler) is calculated according to the following equation:

$$f_{\text{out}} = \frac{5 \text{ MHz}}{2^{\text{prescale}} \cdot (\text{low} + \text{high})}$$

The values for low and high-time stand for the amount of impulses after the prescaler that the internal counter has to reach in order to change the logical level.

The lowest output frequency at a still definable duty cycle of approx. 0...100% is about 0.6Hz.

The highest output frequency where the duty cycle can be still defined in 1%-steps, is 50kHz.

### See also

[PWM\\_ENABLE](#), [PWM\\_OUT](#)

### To be used for the modules

PWM-4(-I)

### Example

```
#INCLUDE ADwinPRO_ALL.inc  
  
INIT:  
  PWM_ENABLE(1,3)           'Enable PWM outputs 1 and 2 for  
                              'output  
  PWM_SET(1,1,0,2500,2500)  'Set PWM signal for output 1  
  PWM_OUT(1,2,1)           'Set PWM output 2 to the logical  
                              'value "1"
```



**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
SSI_MODE(module,pattern)
```

## Parameters

<b>module</b>	Specified module address (1...255).	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. The instruction **SSI\_START** is not necessary for this.

Using the "continuous" mode, some encoder types occasionally return the wrong counter value 0 (zero) instead of the correct counter value. This error does not occur with the "single shot" mode.

## See also

[SSI\\_READ](#), [SSI\\_SET\\_BITS](#), [SSI\\_SET\\_CLOCK](#), [SSI\\_START](#), [SSI\\_STATUS](#)

## To be used for the modules

CO4

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
  SSI_SET_CLOCK(1,200)      'CLK (clock rate) = 50 kHz
  SSI_MODE(1,3)             'Set continuous-mode
                             '(for both encoders)
  SSI_SET_BITS(1,1,23)      'Amount of encoder bits=23 (encoder 1)
  SSI_SET_BITS(1,2,23)      'Amount of encoder bits=23 (encoder 2)

EVENT:
  PAR_1 = SSI_READ(1,1)     'Read out and display position value
                             '(encoder 1)
  PAR_2 = SSI_READ(1,2)     'Read out and display position value
                             '(encoder 2)
```

## SSI\_MODE

## SSI\_READ

**SSI\_READ** returns the last saved counter value of a specified SSI counter on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = SSI_READ(module, dcd_r_no)
```

### Parameters

module	Specified module address (1...255).	LONG
dcd_r_no	Number (1, 2) of the SSI decoder whose counter value is to be read.	LONG
ret_val	Last counter value of the SSI counter (= absolute value position of the encoder).	LONG

### Notes

An encoder value is saved when the bits indicated by **SSI\_SET\_BITS** are read.

Always the amount of bits is returned that is set before by the instruction **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

[SSI\\_MODE](#), [SSI\\_SET\\_BITS](#), [SSI\\_SET\\_CLOCK](#), [SSI\\_START](#), [SSI\\_STATUS](#)

### To be used for the modules

CO4

### Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM m, n, y AS LONG

INIT:
    SSI_SET_CLOCK(1, 50)      'CLK (clock rate) = 200 kHz
    SSI_MODE(1, 1)           'Set continuous-mode (encoder 1)
    SSI_SET_BITS(1, 1, 23)   'Amount of encoder bits=23 (encoder 1)

EVENT:
    PAR_1 = SSI_READ(1, 1)   'Read out and display position
                                'value (encoder 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
```



**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

SSI_SET_BITS (module, dcd_r_no, bit_no)
```

## Parameters

module	Specified module address (1...255).	LONG
dcd_r_no	Number (1, 2) of the SSI decoder whose resolution is to be set.	LONG
bit_no	Amount of bits (1...32) of the bits which are to be read for the encoder (corresponds to the encoder resolution).	LONG

## Notes

The resolution (amount of bits) of the SSI encoder should be similar to the amount of bits which are transferred.

It is always expected to get that certain amount of bits for an encoder value that was indicated before by **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

[SSI\\_MODE](#), [SSI\\_READ](#), [SSI\\_SET\\_CLOCK](#), [SSI\\_START](#), [SSI\\_STATUS](#)

## To be used for the modules

CO4

## Example

```
#INCLUDE ADwinPRO_ALL.inc
```

### INIT:

```
SSI_SET_CLOCK (1,50)      'CLK (clock rate) = 200 kHz
SSI_MODE (1,3)            'Set continuous-mode (for both
                           'encoders)

SSI_SET_BITS (1,1,10)     '10 encoder bits for encoder 1
SSI_SET_BITS (1,2,25)     '25 encoder bits for encoder 2
```

### EVENT:

```
PAR_1 = SSI_READ (1,1)    'Read out and display position value
                           '(encoder 1)
PAR_2 = SSI_READ (1,2)    'Read out and display position value
                           '(encoder 2)
```

## SSI\_SET\_BITS



## SSI\_SET\_CLOCK

**SSI\_SET\_CLOCK** sets the clock rate (approx. 40kHz to 1MHz) on the specified module, with which the encoder is clocked.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

SSI_SET_CLOCK(module,prescale)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>prescale</code>	scale factor (10...255) for setting the clock rate according to the equation: Clock rate = 10MHz / <code>prescale</code> .	LONG

### Notes



The setting of the clock rate is always identical for both encoders, which are connected to the module, and cannot be set separately. If necessary, the clock has to consider the clock rate of the slowest encoder.

Scale factors < 10 are automatically corrected to the value 10; from values > 255 only the least significant 8 bits are used as scale factor.

The possible clock frequency depends on the length of the cable, cable type, and the send and receive components of the encoder or decoder. Basically the following rule applies: The higher the clock frequency the shorter the cable length.

### See also

[SSI\\_MODE](#), [SSI\\_READ](#), [SSI\\_SET\\_BITS](#), [SSI\\_START](#), [SSI\\_STATUS](#)

### To be used for the modules

CO4

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
    SSI_SET_CLOCK(1,10)      'CLK (clock rate) = 1 MHz
    SSI_MODE(1,3)           'Set continuous-mode
                             '(for both encoders)
    SSI_SET_BITS(1,1,10)    'Amount of encoder bits=10 (encoder 1)
    SSI_SET_BITS(1,2,25)    'Amount of encoder bits=25 (encoder 2)

EVENT:
    PAR_1 = SSI_READ(1,1)   'Read out and display position value
                             '(encoder 1)
    PAR_2 = SSI_READ(1,2)   'Read out and display position value
                             '(encoder 2)
```

**SSI\_START** starts the reading of one or both SSI encoders on the specified module (only in mode "single shot").

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
SSI_START(module, dcd_r_no)
```

## Parameters

**module** Specified module address (1...255). LONG

**dcd\_r\_no** Bit pattern for selecting the SSI decoders which are LONG to be started:  
 Bit = 0: No function.  
 Bit = 1: Start reading of the SSI decoder.

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 **SSI\_SET\_BITS**.

A complete encoder value is always transferred, even if the operation mode is changing meanwhile.



## See also

[SSI\\_MODE](#), [SSI\\_READ](#), [SSI\\_SET\\_BITS](#), [SSI\\_SET\\_CLOCK](#), [SSI\\_STATUS](#)

## To be used for the modules

CO4

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
  SSI_SET_CLOCK(1,250)    'CLK (clock rate) = approx. 40 kHz
  SSI_MODE(1,0)           'Set single shot-mode
                          '(both counters)
  SSI_SET_BITS(1,1,23)    'Amount of encoder bits=23 (encoder 1)
  SSI_SET_BITS(1,2,23)    'Amount of encoder bits=23 (encoder 2)

EVENT:
  SSI_START(1,3)          'Read position value of encoders 1 & 2
  DO                      'for encoder 1:
  UNTIL (SSI_STATUS(1,1) = 0)
  REM If position value is read completely, then ...
  PAR_1 = SSI_READ(1,1)    'read out and display position value
  DO                      'For encoder 2:
  UNTIL (SSI_STATUS(1,2) = 0)
  REM If position value is read completely, then ...
  PAR_1 = SSI_READ(1,2)    'read out and display position value
```

## SSI\_STATUS

**SSI\_STATUS** returns the current read-status on the specified module for a specified decoder.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = SSI_STATUS (module, dcd_r_no)
```

### Parameters

module	Specified module address (1...255).	LONG
dcd_r_no	Number (1, 2) of the SSI decoder whose status is to be queried.	LONG
ret_val	Read-status of the decoder: 0: Decoder is ready, that is a complete value has been read. 1: Decoder is reading an encoder value.	LONG

### Notes

Use the status query only in the SSI mode "single shot". In the mode "continuous" querying the status is not useful.

### See also

[SSI\\_MODE](#), [SSI\\_READ](#), [SSI\\_SET\\_BITS](#), [SSI\\_SET\\_CLOCK](#), [SSI\\_START](#)

### To be used for the modules

CO4

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
  SSI_SET_CLOCK(1,250)      'CLK (clock rate) = approx. 40 kHz
  SSI_MODE(1,0)             'Set single shot-mode
                             '(both counters)
  SSI_SET_BITS(1,1,23)      'Amount of encoder bits=23 (encoder 1)
  SSI_SET_BITS(1,2,23)      'Amount of encoder bits=23 (encoder 2)

EVENT:
  SSI_START(1,3)            'Read position value of encoders 1 & 2
  DO                        'For encoder 1:
  UNTIL (SSI_STATUS(1,1) = 0)
  REM If position value is read completely, then ...
  PAR_1 = SSI_READ(1,1)      'Read out and display position value
  DO                        'For encoder 2:
  UNTIL (SSI_STATUS(1,2) = 0)
  REM If position value is read completely, then ...
  PAR_1 = SSI_READ(1,2)      'Read out and display position value
```

**COMP\_DIGIN\_WORD** returns the current status of the threshold value comparison for all channels of the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = COMP_DIGIN_WORD(module)
```

## Parameters

<b>module</b>	Specified module address (1...255).	LONG
<b>ret_val</b>	Bit pattern that returns the status of the threshold value comparison of all channels. (See table below). Bit = 0: Measurement value < lower threshold . Bit = 1: Measurement value > upper threshold.	LONG

Bit no..	31...16	15	14	...	1	0
Channel no.	–	16	15	...	2	1

## Notes

The evaluation of the measurement values is made using switching thresholds (hysteresis), which are specified with **COMP\_SET**. The more the threshold values are apart from each other the more stable the status.

The instruction may be used when single-ended analog signals are presented at the inputs.

## See also

[COMP\\_READ](#), [COMP\\_RESET](#), [COMP\\_FIFO\\_SELECT](#), [COMP\\_SET](#), [COMP\\_DIGIN\\_WORD\\_DIFF](#), [COMP\\_FIFO\\_READ](#)

## To be used for the modules

COMP16 Rev. A

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
    COMP_SET(1,3,500,300)    'Set thresholds of channel 3
                              'to +3V and +1V
                              '(with voltage range -2V ... +8.23V)
    COMP_SET(1,4,600,350)    'Set thresholds of channel 4
                              'to +4V and +.51V
                              '(with voltage range -2V ... +8.23V)

EVENT:
    REM Query the comparison status of channel 3
    PAR_1=(COMP_DIGIN_WORD(1) AND 00100b)
    REM Query the comparison status of channel 4
    PAR_2=(COMP_DIGIN_WORD(1) AND 01000b)
```

## COMP\_DIGIN\_WORD

## COMP\_DIGIN\_WORD\_DIFF

**COMP\_DIGIN\_WORD\_DIFF** returns the current status of the threshold value comparison. The comparison is applied to the difference value of 2 channels (differential signal).

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = COMP_DIGIN_WORD_DIFF(module)
```

### Parameters

<b>module</b>	Specified module address (1...255).	LONG
<b>ret_val</b>	Bit pattern that returns the status of the threshold value comparison of the difference values. (For the assignment of the channels see table below). Bit = 0: Difference < lower threshold value. Bit = 1: Difference > upper threshold value.	LONG

Bit no.	31...8	7	6	...	1	0
Channel pair	–	15,16	13,14	...	3,4	1,2

### Notes

The evaluation of the measurement values is made using switching thresholds (hysteresis), which are specified with **COMP\_SET**. The more the threshold values are apart from each other the more stable the status.

The channel pairs are listed in ascending order (1/2, 3/4, ..., 15/16), the difference is calculated as value<sub>1</sub> - value<sub>2</sub>, value<sub>3</sub> - value<sub>4</sub>, ..., value<sub>15</sub> - value<sub>16</sub>. For the comparison the threshold values of the lower channel are used (= odd channel number).

### See also

[COMP\\_READ](#), [COMP\\_RESET](#), [COMP\\_FIFO\\_SELECT](#), [COMP\\_SET](#), [COMP\\_DIGIN\\_WORD](#), [COMP\\_FIFO\\_READ](#)

### To be used for the modules

COMP16 Rev. A

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
    COMP_SET(1,3,500,300)    'Set thresholds of channel 3
                             'to +3V and +1V
                             '(with voltage range -2V ... +8.23V)
    COMP_SET(1,13,600,350)  'Set thresholds of channel 13
                             'to +2.5V and -1.5V
                             '(with voltage range -2V ... +8.23V)

EVENT:
    REM Query the differential comparison status of channel pair 3/4
    PAR_1=(COMP_DIGIN_WORD_DIFF(1) AND 00000010b)
    REM Query the diff. comparison status of channel pair 13/14
    PAR_2=(COMP_DIGIN_WORD_DIFF(1) AND 01000000b)
```



**COMP\_FIFO\_READ** reads the last 2 x 1024 measurement values of a channel pair from the internal FIFO memory and transfers the values to 2 arrays.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
COMP_FIFO_READ(module,array1[],array2[])
```

## Parameters

module	Specified module address (1...255).	LONG
array1[]	Destination array for 1024 measurement values of the channel with the lower channel number.	ARRAY LONG
array2[]	Destination array for 1024 measurement values of the channel with the higher channel number.	ARRAY LONG

## Notes

In the internal FIFO memory always the last 1024 measurement values of 2 channels are stored. You select the channel pair with **COMP\_FIFO\_SELECT**.

The instruction describes the array elements `arrayx[1]...arrayx[1024]` in both destination arrays. The destination arrays must be sufficiently dimensioned. The transferred measurement values range between 0...1023.

During the process of transferring the measurement data to the destination arrays, no new measurement values are written into the memory. This assures that a complete package of measurement values is transferred.

After data transfer new measurement values can automatically be written into the memory.



## See also

[COMP\\_READ](#), [COMP\\_RESET](#), [COMP\\_FIFO\\_SELECT](#), [COMP\\_SET](#), [COMP\\_DIGIN\\_WORD](#), [COMP\\_DIGIN\\_WORD\\_DIFF](#)

## To be used for the modules

Pro-COMP16 Rev. A

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM array1[1024] AS LONG
DIM array2[1024] AS LONG

INIT:
    COMP_FIFO_SELECT(1.2) 'Write values of the channels 5,6 into
                          'the FIFO memory

EVENT:
    COMP_FIFO_READ(1,array1,array2) 'Get 1024 meas. values for both
                                    'channels: channel 5 in array1,
                                    'channel 6 in array2
```

## COMP\_FIFO\_READ

**COMP\_FIFO\_SELECT**

**COMP\_FIFO\_SELECT** determines the channel pair whose data is stored in the internal FIFO memory of the module.

**Syntax**

```
#INCLUDE ADwinPRO_ALL.inc  
COMP_FIFO_SELECT(module, ch_pair)
```

**Parameters**

module	Specified module address (1...255).	LONG
ch_pair	Number (0...7) to determine a channel pair; the table below illustrates the assignment of the channels.	LONG

ch_pair	7	...	1	0
Channel no.	15, 16	...	3, 4	1, 2

**Notes**

Only one channel pair can be selected; it is not possible to determine more or fewer channel pairs.

The last 1024 measurement values of the two selected channels (= 2 × 1024 values) are written into the FIFO memory. The FIFO memory is read out with **COMP\_FIFO\_READ**.

**See also**

[COMP\\_READ](#), [COMP\\_RESET](#), [COMP\\_SET](#), [COMP\\_DIGIN\\_WORD](#), [COMP\\_DIGIN\\_WORD\\_DIFF](#), [COMP\\_FIFO\\_READ](#)

**To be used for the modules**

COMP16 Rev. A

**Example**

```
#INCLUDE ADwinPRO_ALL.inc  
DIM array1[1024] AS LONG  
DIM array2[1024] AS LONG  
  
INIT:  
    COMP_FIFO_SELECT(1.2)      'Write values of the channels 5,6 into  
                                'the FIFO memory  
  
EVENT:  
    COMP_FIFO_READ(1, array1, array2) 'Get 1024 meas. values for both  
                                        'channels: channel 5 in array1,  
                                        'channel 6 in array2
```

**COMP\_READ** returns the current minimum or maximum measurement value of a specified channel on the module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = COMP_READ(module, channel, value)
```

## Parameters

module	Specified module address (1...255).	LONG
channel	Channel number (1...16).	LONG
value	New status for the selected output: 0: Current measurement value. 1: Minimum measurement value. 2: Maximum measurement value.	LONG
ret_val	Measurement value (0...1023) of the selected channel in digits.	LONG

## Notes

A return value 0 digits corresponds to the minimum analog signal  $U_{\min}$ , the value 1023 digits corresponds to the maximum analog signal  $U_{\max}$ . To convert digits into Volt the following formula applies:

$$\text{Volt} = \text{Digits} \cdot \frac{U_{\max} - U_{\min}}{1023} + U_{\min}$$

This function will not delay or interrupt the measurement of the analog signal.

The minimum and maximum measurement values refer to the measurement values during the time period of the last reset with **COMP\_RESET** (or to the moment of powering up the system).

## See also

[COMP\\_RESET](#), [COMP\\_FIFO\\_SELECT](#), [COMP\\_SET](#), [COMP\\_DIGIN\\_WORD](#), [COMP\\_DIGIN\\_WORD\\_DIFF](#), [COMP\\_FIFO\\_READ](#)

## To be used for the modules

COMP16 Rev. A

## Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
  COMP_SET(1, 3, 500, 300)    'Set thresholds of channel 3
                              'to +3V and +1V
                              '(with voltage range -2V ... +8.23V)
  COMP_RESET(1, 100b)        'Reset the minimum and maximum values
                              'of channel 3

EVENT:
  PAR_1=COMP_READ(1, 3, 1)    'Read minimum of channel 3
  PAR_2=COMP_READ(1, 3, 2)    'Read maximum of channel 3
```

## COMP\_READ

### Conversion digits into Volt

**COMP\_RESET**

**COMP\_RESET** resets the measurement of the minimum and maximum values simultaneously for the selected channels.

**Syntax**

```
#INCLUDE ADwinPRO_ALL.inc  
COMP_RESET(module,pattern)
```

**Parameters**

<code>module</code>	Specified module address (1...255).	LONG
<code>pattern</code>	Bit pattern for resetting the minimum and maximum values (For the assignment of the channels, see table below). Bit = 0: Keep minimum and maximum values. Bit = 1: Reset minimum and maximum values.	LONG

Bit no.	31...16	15	14	...	1	0
Channel no.	–	16	15	...	2	1

**Notes**

During reset the maximum value is set to the value 0, the minimum value to the value 1023. After reset the measurement of both values continues.

**See also**

[COMP\\_READ](#), [COMP\\_FIFO\\_SELECT](#), [COMP\\_SET](#), [COMP\\_DIGIN\\_WORD](#), [COMP\\_DIGIN\\_WORD\\_DIFF](#), [COMP\\_FIFO\\_READ](#)

**To be used for the modules**

COMP16 Rev. A

**Example**

```
#INCLUDE ADwinPRO_ALL.inc
```

**INIT:**

```
REM Set thresholds of channels 1, 3 and 4 to +3V and +1V  
REM (with voltage range -2V ... +8.23V)  
COMP_SET(1,1,500,300)  
COMP_SET(1,3,500,300)  
COMP_SET(1,4,500,300)  
COMP_RESET(1,1101b)      'Reset the minimum and maximum values  
                           'of the channels 1,3,4
```

**EVENT:**

```
PAR_1 = COMP_READ(1,1,1) 'read minimum of channel 1  
PAR_2 = COMP_READ(1,1,2) 'read maximum of channel 1  
PAR_3 = COMP_READ(1,3,1) 'read minimum of channel 3  
PAR_4 = COMP_READ(1,4,2) 'read maximum of channel 4
```

**COMP\_SET** determines the lower and upper threshold value for a specified channel.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

COMP_SET(module, channel, ValHigh, ValLow)
```

## Parameters

module	Specified module address (1...255).	LONG
channel	Channel number(1...16).	LONG
ValHigh	Upper threshold value (1...1023) of the channel.	LONG
ValLow	Lower threshold value (0...1022) of the channel.	LONG

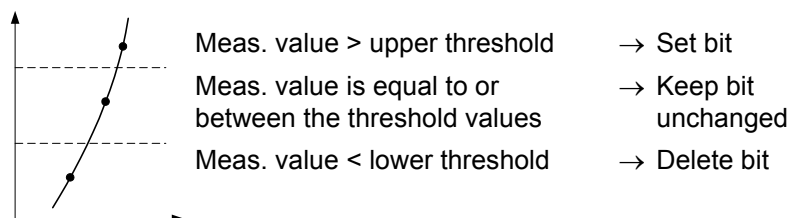
## Notes

When an analog signal is evaluated you define a hysteresis by determining the threshold values.

Example: As long as there is a noisy analog signal near the switching point, the switching status (without hysteresis) changes more or less randomly. If the hysteresis threshold values are selected reasonably the switching status can be stabilized.

The upper threshold value must always be higher than the lower threshold value, otherwise the evaluation of the analog signals is not correct.

The analog signals (parallel on all channels) are measured with a fixed measurement rate (20MHz per channel). Each measurement value is compared with the upper and lower threshold value of the corresponding channel:



The results of the comparison of all channels are saved in a 16-bit word (1 bit for each channel); the current result of the comparison is read out with the instruction **COMP\_DIGIN\_WORD**.

In the same way, the difference value of 2 channels is compared with the threshold values and saved (1 bit for each of the channel pairs). The difference value is calculated as  $value_1 - value_2$ ,  $value_3 - value_4$ , ...,  $value_{15} - value_{16}$ ; the threshold values of the channel with the odd number are used.

The result of the comparison is read out with the instruction **COMP\_DIGIN\_WORD\_DIFF**.

## See also

[COMP\\_READ](#), [COMP\\_RESET](#), [COMP\\_FIFO\\_SELECT](#), [COMP\\_DIGIN\\_WORD](#), [COMP\\_DIGIN\\_WORD\\_DIFF](#), [COMP\\_FIFO\\_READ](#)

## To be used for the modules

COMP16 Rev. A

## COMP\_SET

## Example

```
#INCLUDE ADwinPRO_ALL.inc
```

```
INIT:
```

```
    COMP_SET(1,3,500,300)    'Set threshold values of channel 3  
                             'to +3V and +1V  
                             '(with voltage range -2V ... +8.23V)
```

**RTC\_SET** sets date and time on the real-time clock of the specified module. Invalid values are not accepted.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
```

```
RTC_SET(module, year, month, day, hour, minute, second)
```

## Parameters

module	Specified module address (1...255).	LONG
year	Year (0...63), corresponds to 2000...2063.	LONG
month	Month (1...12).	LONG
day	Day (1...31); valid value ranges according to month and leap year.	LONG
hour	Hour (0...23).	LONG
minute	Minute (0...59).	LONG
second	Second (0...59).	LONG

## Notes

The year refers to the time interval 2000...2063.

## See also

[RTC\\_GET](#)

## To be used for the modules

Storage Rev. A

## Example

```
#INCLUDE ADwinPRO_ALL.inc
```

```
INIT:
```

```
REM Set real-time clock to 4.7.2003 9:17:30
```

```
RTC_SET(1, 3, 7, 4, 9, 17, 30) 'on module 1
```

## RTC\_SET

## RTC\_GET

**RTC\_GET** returns date and time from the real-time clock of the specified module. Invalid values are not accepted.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
```

```
RTC_GET(module, year, month, day, hour, minute, second)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>year</code>	Year (0...63), corresponds to 2000...2063.	LONG
<code>month</code>	Month (1...12).	LONG
<code>day</code>	Day (1...31); valid value ranges according to month and leap year.	LONG
<code>hour</code>	Hour (0...23).	LONG
<code>minute</code>	Minute (0...59).	LONG
<code>second</code>	Second (0...59).	LONG

### Notes

All parameters (except `module`) are return values.

This instruction replaces the following instructions: RTC\_GET\_YEAR, RTC\_GET\_MONTH, RTC\_GET\_DAY, RTC\_GET\_HOUR, RTC\_GET\_MINUTE, RTC\_GET\_SECOND.

### See also

[RTC\\_SET](#)

### To be used for the modules

Storage Rev. A

### Example

```
#INCLUDE ADwinPRO_ALL.inc
```

```
DIM year, mon, day, h, m, s AS LONG
```

```
INIT:
```

```
REM Read real-time clock of module 1
```

```
RTC_GET(1, year, mon, day, h, m, s)
```



**MEDIA\_WR\_BLK\_L** copies a number of LONG data blocks from an array into one file on the storage medium in the specified module.

The first sector into which data is written is individually selectable.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = MEDIA_WR_BLK_L(module, f_info[], file,
                        sec_idx, sec_cnt, array[], array_idx)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>f_info[]</code>	Array, which contains the numbers of the start and end sectors of all files. .	ARRAY LONG
<code>file</code>	Number (1...10) of the file.	LONG
<code>sec_idx</code>	Index (1...m) of the first sector into which is written, referring to the start sector of the file. The max. value depends on the file size. .	LONG
<code>sec_cnt</code>	Amount (1...12) of the data blocks to be transferred (= sectors) à 128 values.	LONG
<code>array[]</code>	Array whose data is transferred.	ARRAY LONG
<code>array_idx</code>	Index (1...n) of the first array element to be transferred.	LONG
<code>ret_val</code>	Status of the glue-logic as bit pattern (see table).	LONG

Bit no.	31:08	07	06	05	04	03	02	01	00
	–	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>

If bit = 1 (bit = 0: without meaning):  
A<sub>1</sub> A storage medium is in the slot.  
A<sub>2</sub> Unknown error occurred.  
A<sub>3</sub> Glue-logic is busy.  
A<sub>4</sub> Data can be read.  
A<sub>5</sub> Data is written.  
A<sub>6</sub> Reset is just being executed.  
A<sub>7</sub> End of file exceeded, no data is transferred.  
A<sub>8</sub> Time out, media did not respond.  
Other bits may be set but are not utilizable here.

## Notes

Before using this instruction, the array `f_info[]` must be initialized with **MEDIA\_RD\_FILEINFO**.

The instruction may only be used in low priority process section:

- anywhere in a low priority process
- in the sections **LOWINIT** or **FINISH** of a high priority process

The start sector is individually selectable. Data is added to already stored data. Do not append more data blocks than the file can receive. Otherwise, the data blocks are not stored!

Each data block contains 128 LONG or FLOAT values and is stored in a sector. For example, if n data blocks are stored, beginning at start sector x, further data blocks are added by writing them into the sector beginning at sector x+n+1.

## MEDIA\_WR\_BLK\_L



The array `array[]` must at least contain `array_idx+s_cnt×128` values.

**See also**

[MEDIA\\_WR\\_BLK\\_F](#), [MEDIA\\_RD\\_BLK\\_L](#), [MEDIA\\_RD\\_BLK\\_F](#),  
[MEDIA\\_RD\\_FILEINFO](#)

**To be used for the modules**

Storage Rev. A

## Example

```

REM #####
REM Save sine table with 3600 points in a file
REM #####
#INCLUDE ADwinPRO_ALL.inc

#DEFINE pstom 1           'address of Pro-STORAGE module
#DEFINE f_info DATA_197  'array with file information
#DEFINE LUT DATA_1       'Look-Up table for sine
#DEFINE file 1            'File no. for saving the sine
#DEFINE nds 3600          'No. of points
REM the array length with the sine data must be a multiple
REM of 128 and greater or equal "nds":
#DEFINE lng 3712          'here: 29*128
#DEFINE pi2 6.2831853     'value of 2*pi

DIM f_info[22] AS LONG AT DM_LOCAL 'array dimensioning
DIM LUT[lng] AS LONG             'Look-Up table with LONG
DIM sec_p[128] AS LONG           'sector for write pointer
DIM idx, n, ret AS LONG         'local variables
DIM sec_s, sec_e, sec_c, sec_n AS LONG 'sector variables
DIM sptr_a, sptr_b AS LONG      'pointer variables (Sector-#)

INIT:
  PAR_52 = MEDIA_RD_FILEINFO(pstom,f_info) 'read file info
  REM check if medium is present and ready for read.
  REM Else -> EXIT
  IF ((PAR_52 AND 1001b) <> 1001b) THEN EXIT

  sec_s = f_info[file*2 + 1] 'First and ...
  sec_e = f_info[file*2 + 2] 'last sector of data file
  sec_c = 1                  'Current (rel.) sector is
                             '1. file sector
  sec_n = SHIFT_RIGHT(nds,7) 'No. of complete sectors of the
  IF ((sec_n*128) < nds) THEN 'table, ... plus one sector,
    INC sec_n                  'if already begun
  ENDIF
  IF ((sec_s+sec_n-1) > sec_e) THEN 'Table greater than file?
    EXIT                      ' then EXIT
  ENDIF

  FOR idx = 1 TO lng          'Calculate sine values
    IF (idx <= nds) THEN
      LUT[idx] = 32767.5 * SIN((idx-1) * pi2 / nds)
    ELSE
      LUT[idx] = 0              'fill rest with 0
    ENDIF
  NEXT idx

  REM write all data sectors
  FOR n=1 TO sec_n
    REM write required sectors individually
    ret = MEDIA_WR_BLK_L(pstom,f_info,file,sec_c,1,LUT,
      (128*n - 127))
    INC sec_c
  NEXT n
  REM 1. sector, where write pointer is saved
  sptr_a = f_info[1] + file - 1
  REM 10. sector, where write pointer duplicate is saved
  sptr_b = sptr_a + 10
  sec_p[1] = nds          '1. LONG in sector = pointer
  sec_p[2] = NOT(nds)      '2. LONG in sector = /pointer
  FOR idx = 3 TO 128      'fill rest (126 LONG) with 0
    sec_p[idx] = 0
  NEXT idx

```

```
REM write both pointer sectors
ret = MEDIA_WR_BLK_L(pstom,f_info,0,file,1,sec_p,1)
ret = MEDIA_WR_BLK_L(pstom,f_info,0,file+10,1,sec_p,1)
```

**MEDIA\_WR\_BLK\_F** copies a number of FLOAT data blocks from an array into one file on the storage medium in the specified module.

The first sector into which data is written is individually selectable.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = MEDIA_WR_BLK_F(module,f_info[],file,sec_
idx,

sec_cnt,array[],array_idx)
```

## Parameters

module	Specified module address (1...255).	LONG
f_info[]	Array, which contains the numbers of the start and end sectors of all files. .	ARRAY LONG
file	Number (1...10) of the file.	LONG
sec_idx	Index (1...m) of the first sector into which is written, referring to the start sector of the file. The max. value depends on the file size. .	LONG
sec_cnt	Amount (1...12) of the data blocks to be transferred (= sectors) à 128 values.	LONG
array[]	Array whose data is transferred.	ARRAY FLOAT
array_idx	Index (1...n) of the first array element to be transferred.	LONG
ret_val	Status of the glue-logic as bit pattern (see table).	LONG

Bit no.	31:08	07	06	05	04	03	02	01	00
	–	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>

If bit = 1 (bit = 0: without meaning):  
A<sub>1</sub> A storage medium is in the slot.  
A<sub>2</sub> Unknown error occurred.  
A<sub>3</sub> Glue-logic is busy.  
A<sub>4</sub> Data can be read.  
A<sub>5</sub> Data is written.  
A<sub>6</sub> Reset is just being executed.  
A<sub>7</sub> End of file exceeded, no data is transferred.  
A<sub>8</sub> Time out, media did not respond.  
Other bits may be set but are not utilizable here.

## Notes

Before using this instruction, the array `f_info[]` must be initialized with **MEDIA\_RD\_FILEINFO**.

The instruction may only be used in low priority process section:

- anywhere in a low priority process
- in the sections **LOWINIT** or **FINISH** of a high priority process

The start sector is individually selectable. Data is added to already stored data. Do not append more data blocks than the file can receive. Otherwise, the data blocks are not stored!

Each data block contains 128 LONG or FLOAT values and is stored in a sector. For example, if n data blocks are stored, beginning at start sec-

## MEDIA\_WR\_BLK\_F



for x, further data blocks are added by writing them into the sector beginning at sector x+n+1.

The array `array[]` must at least contain `array_idx+s_cnt×128` values.

**See also**

[MEDIA\\_RD\\_BLK\\_L](#), [MEDIA\\_RD\\_BLK\\_F](#), [MEDIA\\_RD\\_FILEINFO](#)

**To be used for the modules**

Storage Rev. A

**MEDIA\_RD\_BLK\_L** copies an amount of data blocks with LONG values from one file of the storage medium in the specified module to an array. The first sector to read is individually selectable.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = MEDIA_RD_BLK_L(module, f_info[], file,
                        sec_idx, sec_cnt, array[], array_idx)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>f_info[]</code>	Array which contains the numbers of the start and end sectors of all files. .	ARRAY LONG
<code>file</code>	Number (1...10) of the file.	LONG
<code>sec_idx</code>	Index (1...m) of the first sector into which is written, referring to the start sector of the file. The max. value depends on the file size. .	LONG
<code>sec_cnt</code>	Amount (1...12) of the data blocks to be read (=sectors) à 128 LONG values.	LONG
<code>array[]</code>	Destination array, into which the data is transferred.	ARRAY LONG
<code>array_idx</code>	Index (1...n) of the first array element into which is written.	LONG
<code>ret_val</code>	Status of the glue-logic as bit pattern (see below).	LONG

Bit no.	31:08	07	06	05	04	03	02	01	00
	–	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>

If bit = 1 (bit = 0: without meaning):  
A<sub>1</sub> A storage medium is in the slot.  
A<sub>2</sub> Unknown error occurred.  
A<sub>3</sub> Glue-logic is busy.  
A<sub>4</sub> Data can be read.  
A<sub>5</sub> Data is written.  
A<sub>6</sub> Reset is just being executed.  
A<sub>7</sub> End of file exceeded, no data is transferred.  
A<sub>8</sub> Time out, media did not respond.  
Other bits may be set but are not utilizable here.

## Notes

Before using this instruction, the array `f_info[]` must be initialized with **MEDIA\_RD\_FILEINFO**.

The instruction may only be used in low priority process section:

- anywhere in a low priority process
- in the sections **LOWINIT** or **FINISH** of a high priority process

The start sector is individually selectable. Data is added to already stored data. Do not append more data blocks than the file can receive. Otherwise, the data blocks are not stored!

Each data block contains 128 LONG values and is stored in a sector. The destination array `array[]` must at least contain `array_idx+sec_cnt×128` array elements.

## MEDIA\_RD\_BLK\_L



**See also**

[MEDIA\\_WR\\_BLK\\_L](#), [MEDIA\\_WR\\_BLK\\_F](#), [MEDIA\\_RD\\_BLK\\_F](#),  
[MEDIA\\_RD\\_FILEINFO](#)

**To be used for the modules**

Storage Rev. A



## Example

```

REM #####
REM read sine table with 3600 points from data file
REM #####
#INCLUDE ADwinPRO_ALL.inc

#DEFINE pstom 1           'address of Pro-STORAGE module
#DEFINE f_info DATA_197  'array with file information
#DEFINE LUT DATA_1       'Look-Up table for sine
#DEFINE file 1            'File no. for reading the sine
#DEFINE nds 3600          'No. of points
REM the array length with the sine data must be a multiple
REM of 128 and greater or equal "nds":
#DEFINE lng 3712          'here: 29*128
#DEFINE pi2 6.2831853     'value of 2*pi

DIM f_info[22] AS LONG AT DM_LOCAL 'array dimensioning
DIM LUT[lng] AS LONG              'Look-Up table with LONG
DIM sec_p[128] AS LONG            'sector for write pointer
DIM idx, n, ret AS LONG           'local variables
DIM sec_c, sec_n AS LONG          'sector variables
DIM dzn, rmn AS LONG              '12 sector blocks, add. values

INIT:
  PAR_52 = MEDIA_RD_FILEINFO(pstom,f_info) 'read file info
  REM check if medium is present and ready for read.
  REM Else -> EXIT
  IF ((PAR_52 AND 1001b) <> 1001b) THEN EXIT

  REM get length of saved table: read 1. pointer sector
  ret = MEDIA_RD_BLK_L(pstom,f_info,(file-1),1,1,sec_p,1)
  IF ((ret AND 22h) > 0) THEN EXIT 'Exit if Reset o. Error

  REM check validity of 1. data pointer
  IF ((sec_p[1] XOR sec_p[2]) <> -1) THEN

    REM 1. data pointer is invalid -> read 2. data pointer
    ret = MEDIA_RD_BLK_L(pstom,f_info,(file-1),11,1,sec_p,1)
    IF ((ret AND 22h) > 0) THEN EXIT 'Exit bei RESET o. ERROR

    REM check validity of 2. data pointer
    IF ((sec_p[1] XOR sec_p[2]) <> -1) THEN
      REM 2. data pointer is invalid, too -> Exit
      EXIT
    ELSE
      nds = sec_p[1]           'use 2. data pointer
    ENDIF
  ELSE
    nds = sec_p[1]           'use 1. data pointer
  ENDIF

  REM calculate no. of sectors and packets (12 sectors) to read
  dzn = nds/1536              'No. of packets, 12 sectors each
  rmn = nds - dzn*1536        'No. of remaining LONGs
  sec_n = SHIFT_RIGHT(rmn,7)  'No. further sectors to read
  IF ((128*sec_n) < rmn) THEN
    REM sector was begun: add one
    INC sec_n
  ENDIF

  REM copy sectors of data file into array
  IF (dzn>0) THEN             'any 12-sector packets?
    FOR n=1 TO dzn            'copy all packets ...
      sec_c = 12*n - 11        'calc. current sector
      idx = 1536*n - 1535      'calc. pointer in dest. array
    
```

```
REM read 12 sectors
ret = MEDIA_RD_BLK_L(pstom,f_info,file,sec_c,12,LUT,idx)
IF ((ret AND 22h) > 0) THEN EXIT 'Exit if Reset or Error
NEXT n
ENDIF

REM copy remaining sectores
ret = MEDIA_RD_BLK_L(pstom,f_info,file,(12*dzn+1),sec,
n,LUT,(1536*dzn+1))
IF ((ret AND 22h) > 0) THEN EXIT 'Exit if Reset or Error
```

**MEDIA\_RD\_BLK\_F** copies an amount of data blocks with FLOAT values from one file of the storage medium in the specified module to an array. The first sector to be read of the file is individually selectable.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = MEDIA_RD_BLK_F(module,f_info[],file,
                        sec_idx,s_cnt,array[],array_idx)
```

## Parameters

module	Specified module address (1...255).	LONG
f_info[]	Array which contains the numbers of the start and end sectors of all files. .	ARRAY LONG
file	Number (1...10) of the file.	LONG
sec_idx	Index (1...m) of the first sector into which is written, referring to the start sector of the file. The max. value depends on the file size. .	LONG
sec_cnt	Amount (1...12) of the data blocks to be read (= sectors) à 128 LONG values.	LONG
array[]	Destination array, into which the data is transferred.	ARRAY LONG
array_idx	Index (1...n) of the first array element into which is written.	LONG
ret_val	Status of the glue-logic as bit pattern (see below).	LONG

Bit no.	31:08	07	06	05	04	03	02	01	00
	–	A <sub>8</sub>	A <sub>7</sub>	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>

If bit = 1 (bit = 0: without meaning):  
A<sub>1</sub> A storage medium is in the slot.  
A<sub>2</sub> Unknown error occurred.  
A<sub>3</sub> Glue-logic is busy.  
A<sub>4</sub> Data can be read.  
A<sub>5</sub> Data is written.  
A<sub>6</sub> Reset is just being executed.  
A<sub>7</sub> End of file exceeded, no data is transferred.  
A<sub>8</sub> Time out, media did not respond.  
Other bits may be set but are not utilizable here.

## Notes

Before using this instruction, the array `f_info[]` must be initialized with **MEDIA\_RD\_FILEINFO**.

The instruction may only be used in low priority process section:

- anywhere in a low priority process
- in the sections **LOWINIT** or **FINISH** of a high priority process

The start sector is individually selectable. Data is added to already stored data. Do not append more data blocks than the file can receive. Otherwise, the data blocks are not stored!

Each data block contains 128 LONG values and is stored in a sector. The destination array `array[]` must at least contain a number of `array_idx+s_cnt*128` array elements.

## MEDIA\_RD\_BLK\_F



**See**

[MEDIA\\_WR\\_BLK\\_L](#), [MEDIA\\_WR\\_BLK\\_F](#), [MEDIA\\_RD\\_BLK\\_L](#),  
[MEDIA\\_RD\\_FILEINFO](#)

**To be used for the modules**

Storage Rev. A

**MEDIA\_RD\_FILEINFO** initializes the glue-logic on the specified module and returns the file information (start and end sector) into an array.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = MEDIA_RD_FILEINFO(module, f_info[])
```

## Parameters

module	Specified module address (1...255).	LONG
f_info[]	Array which contains the numbers of the start and end sectors of all files.	ARRAY LONG
ret_val	Status of the glue-logic as bit pattern (see table).	LONG

Bit No.	31:06	05	04	03	02	01	00
	—	A <sub>6</sub>	A <sub>5</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>

If bit = 1 (bit = 0: without meaning):

A<sub>1</sub> A storage medium is in the slot.

A<sub>2</sub> Unknown error occurred.

A<sub>3</sub> Glue-logic is busy.

A<sub>4</sub> Data can be read.

A<sub>5</sub> Data is written.

A<sub>6</sub> Reset is just being executed.

Other bits may be set but are not utilizable here.

## Notes

The array `f_info[]` must be initialized with **MEDIA\_RD\_FILEINFO**, before data is written to or read from the storage medium. The array must have the size of at least 22 elements.

The instruction may only be used in low priority process section:

- anywhere in a low priority process
- in the sections **LOWINIT** or **FINISH** of a high priority process

The array elements with odd-numbered index contain the start sector of a file, those with even-numbered index contain the end sectors. The following table shows the assignment between index numbers of the array and the files.

Index No.	File
1, 2	Fileinfo.dat
3, 4	ADWIN1.dat
...	...
21, 22	ADWIN10.dat

## See also

[MEDIA\\_WR\\_BLK\\_L](#), [MEDIA\\_WR\\_BLK\\_F](#), [MEDIA\\_RD\\_BLK\\_L](#), [MEDIA\\_RD\\_BLK\\_F](#)

## To be used for the modules

Storage Rev. A

## MEDIA\_RD\_FILEINFO



## Example

```

REM #####
REM read and check the file <FILEINFO.DAT>
REM #####
REM PAR_1 ... PAR_10: First sector of data file 1 ... 10
REM PAR_11 ... PAR_20: Last sector of data file 1 ... 10
REM PAR_21 ... PAR_30: No. of sectors in data file 1 ... 10
REM PAR_31 ... PAR_40: No. of values in data file 1 ... 10
REM #####
#INCLUDE ADwinPRO_ALL.inc

#DEFINE pstom 1                'address of Pro-STORAGE module
#DEFINE f_info DATA_197      'array with file information

DIM f_info[22] AS LONG AT DM_LOCAL 'array dimensioning
DIM n AS LONG                'local variables

INIT:
    PAR_52 = MEDIA_RD_FILEINFO(pstom,f_info) 'read file info
    REM check if medium is present and ready for read.
    REM Else -> EXIT
    IF ((PAR_52 AND 1001b) <> 1001b) THEN EXIT

    REM read all 10 files
    FOR n = 1 TO 10
        REM Calc. no. of first and last file sector
        PAR[n] = f_info[n*2 + 1]
        PAR[n+10] = f_info[n*2 + 2]
        REM file length > 1 sector?
        IF ((PAR[n+10] - PAR[n]) <> 0) THEN
            PAR[n+20] = PAR[n+10] - PAR[n] + 1 'No. of sectors
            PAR[n+30] = PAR[n+20] * 128 'No. of values
        ENDIF
    NEXT n

EXIT

```

### 3.5 Pro I: Special Modules

This section describes instructions necessary to access special *ADwin-Pro* modules. To use these instructions insert this line in your *ADbasic* program:

```
#INCLUDE ADwinPRO_ALL.inc
```

On the following pages the instructions will be described more detailed, including an easy application example. The instructions are sorted by type of interface:

- [Thermocouples, page 163](#)
- [CAN bus, page 179](#)
- [Field bus, page 194](#)
- [RSxxx, page 204](#)
- [LS-Bus + Pro I, page 213](#)

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

#### 3.5.1 Thermocouples

## PT100\_DIG\_TO\_TEMP

**PT100\_DIG\_TO\_TEMP** calculates the temperature in degrees Celsius or Fahrenheit from the measured digital value of a PT100 sensor.

### Syntax

```
#INCLUDE ADWINPRO_ALL.INC  
  
ret_val = PT100_DIG_TO_TEMP(dig_val, t_unit)
```

### Parameters

dig_val	Digital value (0...65535).	LONG
t_unit	Temperature unit: 1: degree Celsius. 2: degree Fahrenheit.	LONG
ret_val	Temperature in degree Celsius oder Fahrenheit.	FLOAT

### Notes

The thermoelectric voltage and the temperature values in °C and °F apply only for standard PT100 sensors according to the norm IEC 751.

Alternatively, the temperature may be calculated manually using a conversion table: Thermocouple-amplifier PT100-x.

You will find these tables in the *ADbasic* online help, menu entry "Hardware information" (Contents) or in the data sheet of the manufacturer: Analog Devices.

### See also

[TC\\_SELECT](#), [PT100\\_DIG\\_TO\\_R](#)

### To be used for the modules

PT100-4, PT100-8

### Example

It is presumed in the example that the analog output of the PT100 module is connected with the analog input 1 of an A/D module with the module address 1.

```
#INCLUDE ADWINPRO_ALL.INC  
DIM temp[8] AS FLOAT  
DIM channel AS LONG  
  
INIT:  
PROCESSDELAY=100000  
  
EVENT:  
FOR channel = 1 TO 8  
  TC_SELECT(1,channel) 'select next channel  
  P1_SLEEP(500) 'wait for settling time  
  REM read thermoelectric voltage and convert into °C  
  temp[channel] = PT100_DIG_TO_TEMP(ADC(1,1),1)  
NEXT
```



**PT100\_DIG\_TO\_R** calculates the resistance in Ohm from the measured digital value of a PT100 sensor.

## Syntax

```
#INCLUDE ADWINPRO_ALL.INC

ret_val = PT100_DIG_TO_R(dig_val)
```

## Parameters

dig_val	Digital value (0...65535).	LONG
ret_val	Resistance in Ohm.	FLOAT

## Notes

The calculation of the resistance values from the thermoelectric voltage uses the following formula:

$$\text{ret\_val} = ((\text{dig\_val} - 32768) \cdot 200 / 65536) + 100\Omega$$

## See also

[TC\\_SELECT](#), [PT100\\_DIG\\_TO\\_TEMP](#)

## To be used for the modules

PT100-4, PT100-8

## Example

It is presumed in the example that the analog output of the thermocouple module is connected with the analog input 1 of an A/D module with the module address 1.

```
#INCLUDE ADWINPRO_ALL.INC
DIM temp[8] AS FLOAT
DIM channel AS LONG

INIT:
PROCESSDELAY=100000

EVENT:
FOR channel = 1 TO 8
TC_SELECT(1,channel) 'select next channel
P1_SLEEP(500) 'wait for settling time
REM read thermoelectric voltage and convert into °C
temp[channel] = PT100_DIG_TO_R(ADC(1,1),1)
NEXT
```

## PT100\_DIG\_TO\_R

## TC\_SELECT

**TC\_SELECT** sets the thermocouple channels via multiplexer to the analog output of the module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
TC_SELECT(module, channel)
```

### Parameters

module	Specified module address (1...255).	LONG
channel	Channel number: TC-4, PT100-4: 1...4. TC-8, PT100-8: 1...8. TC-16: 1...16.	LONG

### Notes



The multiplexer settling time must be bridged by programming correspondingly, so as to ensure a correct measurement value. The settling time is given in the Pro-Hardware manual.

In order to convert the voltage to [°C], you need the conversion tables of the thermocouple amplifier:

- TC-x Type J (AD594)
- TC-x Type K (AD595)
- PT100-x

You will find these tables in the *ADbasic* online help (header Contents) or in the data sheet of the manufacturer: Analog Devices.

### See also

[PT100\\_DIG\\_TO\\_TEMP](#), [PT100\\_DIG\\_TO\\_R](#), [TCJ\\_DIG\\_TO\\_TEMP](#),  
[TCK\\_DIG\\_TO\\_TEMP](#)

### Can be used for the modules

Pro-TC-4, Pro-TC-8, Pro-TC-16  
Pro-PT100-4, Pro-PT100-8

## Example

It is presumed in the example that the analog output of the thermocouple module is connected with the analog input 1 of an A/D module with the module address 1.

```
#INCLUDE ADwinPRO_ALL.inc
DIM value, i AS LONG

INIT:
    value = 0          'Initialize ...
    i = 1              ' variables

EVENT:
    TC_SELECT(1,3)      'Select thermocouple channel 3
    REM Insert some program code so that the temperature
    REM measurement is done after the multiplexer settling time.
    REM The following instruction ADC already needs a part
    REM of that settling time.
    value = value + ADC(1,1) 'Measure value
    IF (i = 10) THEN
        PAR_1 = value / 10    'Mean value of 10 measurements
        value = 0
        i = 0
    ENDIF
    INC i
```

## TCJ\_DIG\_TO\_TEMP

**TCJ\_DIG\_TO\_TEMP** calculates the temperature in degrees Celsius or Fahrenheit from the measured digital value of a thermo couple type J.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = TCJ_DIG_TO_TEMP(dig_val, t_unit)
```

### Parameters

dig_val	Digital value (0...65535).	LONG
t_unit	Temperature unit: 1: degree Celsius. 2: degree Fahrenheit.	LONG
ret_val	Temperature in degree Celsius or Fahrenheit.	FLOAT

### Notes

The thermoelectric voltage and the temperature values in °C and °F apply only for standard thermocouples type J according to the norm IEC 584-1.

Alternatively, the temperature may be calculated manually using a conversion table: Thermocouple-amplifier TC-x Typ J (AD594).

You will find these tables in the *ADbasic* online help (or in the data sheet of the manufacturer: Analog Devices).

### See also

[TC\\_SELECT](#), [TCJ\\_DIG\\_TO\\_TEMP](#), [TCK\\_DIG\\_TO\\_TEMP](#)

### Can be used for the modules

Pro-TC-4, Pro-TC-8, Pro-TC-16

### Example

It is presumed in the example that the analog output of the thermocouple module is connected with the analog input 1 of an A/D module with the module address 1.

```
#INCLUDE ADWINPRO_ALL.INC
DIM temp[8] AS FLOAT
DIM channel AS LONG

INIT:
    PROCESSDELAY=100000

EVENT:
    FOR channel = 1 TO 8
        TC_SELECT(1,channel)    'select next channel
        P1_SLEEP(500)           'wait for settling time
        REM read thermoelectric voltage and convert into °C
        temp[channel] = TCJ_DIG_TO_TEMP(ADC(1,1),1)
    NEXT
    FPAR_1 = temp[1]            'Temperature channel 1 in °C
    ...
    FPAR_8 = temp[8]            'Temperature channel 8 in °C
```

**TCK\_DIG\_TO\_TEMP** calculates the temperature in degrees Celsius or Fahrenheit from the measured digital value of a thermo couple type K.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = TCK_DIG_TO_TEMP(dig_val, t_unit)
```

## Parameters

dig_val	Digital value (0...65535).	LONG
t_unit	Temperature unit: 1: degree Celsius. 2: degree Fahrenheit.	LONG
ret_val	Temperature in degree Celsius oder Fahrenheit.	FLOAT

## Notes

The thermoelectric voltage and the temperature values in °C and °F apply only for standard thermocouples type J according to the norm IEC 584-1.

Alternatively, the temperature may be calculated manually using a conversion table: Thermocouple-amplifier TC-x Type K (AD595).

You will find these tables in the *ADbasic* online help (or in the data sheet of the manufacturer: Analog Devices).

## See also

[TC\\_SELECT](#), [TCJ\\_DIG\\_TO\\_TEMP](#)

## Can be used for the modules

Pro-TC-4, Pro-TC-8, Pro-TC-16

## Example

It is presumed in the example that the analog output of the thermocouple module is connected with the analog input 1 of an A/D module with the module address 1.

```
#INCLUDE ADWINPRO_ALL.INC
DIM temp[8] AS FLOAT
DIM channel AS LONG

INIT:
    PROCESSDELAY=100000

EVENT:
    FOR channel = 1 TO 8
        TC_SELECT(1,channel) 'select next channel
        P1_SLEEP(500)        'wait for settling time
        REM read thermoelectric voltage and convert into °C
        temp[channel] = TCK_DIG_TO_TEMP(ADC(1,1),1)
    NEXT
    FPAR_1 = temp[1]          'Temperature channel 1 in °C
    ...
    FPAR_8 = temp[8]          'Temperature channel 8 in °C
```

## TCK\_DIG\_TO\_TEMP

## TC\_READ\_B

**TC\_READ\_B** returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$  /  $^{\circ}\text{F}$ ) of a specified channel on the module.

The temperature value is valid for a thermocouple type B only.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = TC_READ_B(module, channel, ret_type)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>channel</code>	Channel number (1...8).	LONG
<code>ret_type</code>	Type of return value <code>ret_val</code> : 0: Thermoelectric voltage in $\mu\text{V}$ . 1: Temperature in $^{\circ}\text{C}$ . 2: Temperature in $^{\circ}\text{F}$ .	LONG
<code>ret_val</code>	Measured value of the channel, refers to <code>ret_type</code> : Thermoelectric voltage: $291\mu\text{V}$ ... $13820\mu\text{V}$ . Temperature in $^{\circ}\text{C}$ : $250^{\circ}\text{C}$ ... $1820^{\circ}\text{C}$ . Temperature in $^{\circ}\text{F}$ : $482^{\circ}\text{F}$ ... $3329.6^{\circ}\text{F}$ .	FLOAT

### Notes

The module samples the temperature regularly (setting of the sample rate see **TC\_SET\_RATE**). The instruction **TC\_READ\_B** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$  apply only for standard thermocouples type E according to the norm IEC 584-1.

### See also

[TC\\_READ\\_E](#), [TC\\_READ\\_J](#), [TC\\_READ\\_K](#), [TC\\_READ\\_N](#), [TC\\_READ\\_R](#), [TC\\_READ\\_S](#), [TC\\_READ\\_T](#), [TC\\_SET\\_RATE](#)

### Can be used for the modules

Pro-TC-8 ISO

### Example

```
#INCLUDE ADwinPRO_ALL.inc

EVENT:
REM Read temperature in  $^{\circ}\text{C}$  from channel 5
FPAR_1 = TC_READ_B(1,5,1)
```

**TC\_READ\_E** returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$  /  $^{\circ}\text{F}$ ) of a specified channel on the module.

The temperature value is valid for a thermocouple type E only.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = TC_READ_E(module, channel, ret_type)
```

## Parameters

module	Specified module address (1...255).	LONG
channel	Channel number (1...8).	LONG
ret_type	Type of return value <code>ret_val</code> : 0: Thermoelectric voltage in $\mu\text{V}$ . 1: Temperature in $^{\circ}\text{C}$ . 2: Temperature in $^{\circ}\text{F}$ .	LONG
ret_val	Measured value of the channel, refers to <code>ret_type</code> : Thermoelectric voltage: $-8825\mu\text{V}$ ... $76373\mu\text{V}$ . Temperature in $^{\circ}\text{C}$ : $-200^{\circ}\text{C}$ ... $1000^{\circ}\text{C}$ . Temperature in $^{\circ}\text{F}$ : $-328^{\circ}\text{F}$ ... $1832^{\circ}\text{F}$ .	FLOAT

## Notes

The module samples the temperature regularly (setting of the sample rate see **TC\_SET\_RATE**). The instruction **TC\_READ\_E** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$  apply only for standard thermocouples type E according to the norm IEC 584-1.

## See also

[TC\\_READ\\_B](#), [TC\\_READ\\_J](#), [TC\\_READ\\_K](#), [TC\\_READ\\_N](#), [TC\\_READ\\_R](#), [TC\\_READ\\_S](#), [TC\\_READ\\_T](#), [TC\\_SET\\_RATE](#)

## Can be used for the modules

Pro-TC-8 ISO

## Example

```
#INCLUDE ADwinPRO_ALL.inc

EVENT:
REM Read temperature in  $^{\circ}\text{C}$  from channel 5
FPAR_1 = TC_READ_E(1,5,1)
```

## TC\_READ\_E

## TC\_READ\_J

**TC\_READ\_J** returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$  /  $^{\circ}\text{F}$ ) of a specified channel on the module.

The temperature value is valid for a thermocouple type J only.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = TC_READ_J(module, channel, ret_type)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>channel</code>	Channel number (1...8).	LONG
<code>ret_type</code>	Type of return value <code>ret_val</code> : 0: Thermoelectric voltage in $\mu\text{V}$ . 1: Temperature in $^{\circ}\text{C}$ . 2: Temperature in $^{\circ}\text{F}$ .	LONG
<code>ret_val</code>	Measured value of the channel, refers to <code>ret_type</code> : Thermoelectric voltage: $-8095\mu\text{V}$ ... $69553\mu\text{V}$ . Temperature in $^{\circ}\text{C}$ : $-210^{\circ}\text{C}$ ... $1200^{\circ}\text{C}$ . Temperature in $^{\circ}\text{F}$ : $-346^{\circ}\text{F}$ ... $2192^{\circ}\text{F}$ .	FLOAT

### Notes

The module samples the temperature regularly (setting of the sample rate see **TC\_SET\_RATE**). The instruction **TC\_READ\_J** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$  apply only for standard thermocouples type J according to the norm IEC 584-1.

### See also

[TC\\_READ\\_B](#), [TC\\_READ\\_E](#), [TC\\_READ\\_K](#), [TC\\_READ\\_N](#), [TC\\_READ\\_R](#), [TC\\_READ\\_S](#), [TC\\_READ\\_T](#), [TC\\_SET\\_RATE](#)

### Can be used for the modules

Pro-TC-8 ISO

### Example

```
#INCLUDE ADwinPRO_ALL.inc

EVENT:
REM Read temperature in  $^{\circ}\text{C}$  from channel 5
FPAR_1 = TC_READ_J(1,5,1)
```



**TC\_READ\_K** returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$  /  $^{\circ}\text{F}$ ) of a specified channel on the module.

The temperature value is valid for a thermocouple type K only.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = TC_READ_K(module, channel, ret_type)
```

## Parameters

module	Specified module address (1...255).	LONG
channel	Channel number (1...8).	LONG
ret_type	Type of return value <code>ret_val</code> : 0: Thermoelectric voltage in $\mu\text{V}$ . 1: Temperature in $^{\circ}\text{C}$ . 2: Temperature in $^{\circ}\text{F}$ .	LONG
ret_val	Measured value of the channel, refers to <code>ret_type</code> : Thermoelectric voltage: $-5891\mu\text{V}$ ... $54886\mu\text{V}$ . Temperature in $^{\circ}\text{C}$ : $-200^{\circ}\text{C}$ ... $1372^{\circ}\text{C}$ . Temperature in $^{\circ}\text{F}$ : $-328^{\circ}\text{F}$ ... $2501.6^{\circ}\text{F}$ .	FLOAT

## Notes

The module samples the temperature regularly (setting of the sample rate see **TC\_SET\_RATE**). The instruction **TC\_READ\_K** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$  apply only for standard thermocouples type K according to the norm IEC 584-1.

## See also

[TC\\_READ\\_B](#), [TC\\_READ\\_E](#), [TC\\_READ\\_J](#), [TC\\_READ\\_N](#), [TC\\_READ\\_R](#), [TC\\_READ\\_S](#), [TC\\_READ\\_T](#), [TC\\_SET\\_RATE](#)

## Can be used for the modules

Pro-TC-8 ISO

## Example

```
#INCLUDE ADwinPRO_ALL.inc

EVENT:
REM Read temperature in  $^{\circ}\text{C}$  from channel 5
FPAR_1 = TC_READ_K(1,5,1)
```

## TC\_READ\_K

## TC\_READ\_N

**TC\_READ\_N** returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$  /  $^{\circ}\text{F}$ ) of a specified channel on the module.

The temperature value is valid for a thermocouple type N only.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = TC_READ_N(module, channel, ret_type)
```

### Parameters

module	Specified module address (1...255).	LONG
channel	Channel number (1...8).	LONG
ret_type	Type of return value <code>ret_val</code> : 0: Thermoelectric voltage in $\mu\text{V}$ . 1: Temperature in $^{\circ}\text{C}$ . 2: Temperature in $^{\circ}\text{F}$ .	LONG
ret_val	Measured value of the channel, refers to <code>ret_type</code> : Thermoelectric voltage: $-3990\mu\text{V}$ ... $47513\mu\text{V}$ . Temperature in $^{\circ}\text{C}$ : $-200^{\circ}\text{C}$ ... $1300^{\circ}\text{C}$ . Temperature in $^{\circ}\text{F}$ : $-328^{\circ}\text{F}$ ... $2372^{\circ}\text{F}$ .	FLOAT

### Notes

The module samples the temperature regularly (setting of the sample rate see **TC\_SET\_RATE**). The instruction **TC\_READ\_N** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$  apply only for standard thermocouples type N according to the norm IEC 584-1.

### See also

[TC\\_READ\\_B](#), [TC\\_READ\\_E](#), [TC\\_READ\\_J](#), [TC\\_READ\\_K](#), [TC\\_READ\\_R](#), [TC\\_READ\\_S](#), [TC\\_READ\\_T](#), [TC\\_SET\\_RATE](#)

### Can be used for the modules

Pro-TC-8 ISO

### Example

```
#INCLUDE ADwinPRO_ALL.inc

EVENT:
    REM Read temperature in  $^{\circ}\text{C}$  from channel 5
    FPAR_1 = TC_READ_N(1,5,1)
```

**TC\_READ\_R** returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$  /  $^{\circ}\text{F}$ ) of a specified channel on the module.

The temperature value is valid for a thermocouple type R only.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = TC_READ_R(module, channel, ret_type)
```

## Parameters

module	Specified module address (1...255).	LONG
channel	Channel number (1...8).	LONG
ret_type	Type of return value <code>ret_val</code> : 0: Thermoelectric voltage in $\mu\text{V}$ . 1: Temperature in $^{\circ}\text{C}$ . 2: Temperature in $^{\circ}\text{F}$ .	LONG
ret_val	Measured value of the channel, refers to <code>ret_type</code> : Thermoelectric voltage: -226 $\mu\text{V}$ ... 21101 $\mu\text{V}$ . Temperature in $^{\circ}\text{C}$ : -50 $^{\circ}\text{C}$ ... 1768 $^{\circ}\text{C}$ . Temperature in $^{\circ}\text{F}$ : -58 $^{\circ}\text{F}$ ... 3214.4 $^{\circ}\text{F}$ .	FLOAT

## Notes

The module samples the temperature regularly (setting of the sample rate see **TC\_SET\_RATE**). The instruction **TC\_READ\_R** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$  apply only for standard thermocouples type R according to the norm IEC 584-1.

## See also

[TC\\_READ\\_B](#), [TC\\_READ\\_E](#), [TC\\_READ\\_J](#), [TC\\_READ\\_K](#), [TC\\_READ\\_N](#), [TC\\_READ\\_S](#), [TC\\_READ\\_T](#), [TC\\_SET\\_RATE](#)

## Can be used for the modules

Pro-TC-8 ISO

## Example

```
#INCLUDE ADwinPRO_ALL.inc

EVENT:
REM Read temperature in  $^{\circ}\text{C}$  from channel 5
FPAR_1 = TC_READ_R(1,5,1)
```

## TC\_READ\_R

## TC\_READ\_S

**TC\_READ\_S** returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$  /  $^{\circ}\text{F}$ ) of a specified channel on the module.

The temperature value is valid for a thermocouple type S only.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = TC_READ_S(module, channel, ret_type)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>channel</code>	Channel number (1...8).	LONG
<code>ret_type</code>	Type of return value <code>ret_val</code> : 0: Thermoelectric voltage in $\mu\text{V}$ . 1: Temperature in $^{\circ}\text{C}$ . 2: Temperature in $^{\circ}\text{F}$ .	LONG
<code>ret_val</code>	Measured value of the channel, refers to <code>ret_type</code> : Thermoelectric voltage: $-236\mu\text{V}$ ... $18693\mu\text{V}$ . Temperature in $^{\circ}\text{C}$ : $-50^{\circ}\text{C}$ ... $1768^{\circ}\text{C}$ . Temperature in $^{\circ}\text{F}$ : $-58^{\circ}\text{F}$ ... $3214.4^{\circ}\text{F}$ .	FLOAT

### Notes

The module samples the temperature regularly (setting of the sample rate see **TC\_SET\_RATE**). The instruction **TC\_READ\_S** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$  apply only for standard thermocouples type S according to the norm IEC 584-1.

### See also

[TC\\_READ\\_B](#), [TC\\_READ\\_E](#), [TC\\_READ\\_J](#), [TC\\_READ\\_K](#), [TC\\_READ\\_N](#), [TC\\_READ\\_R](#), [TC\\_READ\\_T](#), [TC\\_SET\\_RATE](#)

### Can be used for the modules

Pro-TC-8 ISO

### Example

```
#INCLUDE ADwinPRO_ALL.inc

EVENT:
REM Read temperature in  $^{\circ}\text{C}$  from channel 5
FPAR_1 = TC_READ_S(1,5,1)
```

**TC\_READ\_T** returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$  /  $^{\circ}\text{F}$ ) of a specified channel on the module.

The temperature value is valid for a thermocouple type T only.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = TC_READ_T(module, channel, ret_type)
```

## Parameters

module	Specified module address (1...255).	LONG
channel	Channel number (1...8).	LONG
ret_type	Type of return value <code>ret_val</code> : 0: Thermoelectric voltage in $\mu\text{V}$ . 1: Temperature in $^{\circ}\text{C}$ . 2: Temperature in $^{\circ}\text{F}$ .	LONG
ret_val	Measured value of the channel, refers to <code>ret_type</code> : Thermoelectric voltage: $-5603\mu\text{V}$ ... $20872\mu\text{V}$ . Temperature in $^{\circ}\text{C}$ : $-200^{\circ}\text{C}$ ... $400^{\circ}\text{C}$ . Temperature in $^{\circ}\text{F}$ : $-454^{\circ}\text{F}$ ... $752^{\circ}\text{F}$ .	FLOAT

## Notes

The module samples the temperature regularly (setting of the sample rate see **TC\_SET\_RATE**). The instruction **TC\_READ\_T** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in  $^{\circ}\text{C}$  and  $^{\circ}\text{F}$  apply only for standard thermocouples type T according to the norm IEC 584-1.

## See also

[TC\\_READ\\_B](#), [TC\\_READ\\_E](#), [TC\\_READ\\_J](#), [TC\\_READ\\_K](#), [TC\\_READ\\_N](#), [TC\\_READ\\_R](#), [TC\\_READ\\_S](#), [TC\\_SET\\_RATE](#)

## Can be used for the modules

Pro-TC-8 ISO

## Example

```
#INCLUDE ADwinPRO_ALL.inc

EVENT:
REM Read temperature in  $^{\circ}\text{C}$  from channel 5
FPAR_1 = TC_READ_T(1,5,1)
```

## TC\_READ\_T

## TC\_SET\_RATE

**TC\_SET\_RATE** sets the sample rate for the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
TC_SET_RATE(module, rate)
```

### Parameters

<b>module</b>	Specified module address (1...255).	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

[TC\\_READ\\_B](#), [TC\\_READ\\_E](#), [TC\\_READ\\_J](#), [TC\\_READ\\_K](#), [TC\\_READ\\_N](#), [TC\\_READ\\_R](#), [TC\\_READ\\_S](#), [TC\\_READ\\_T](#)

### Can be used for the modules

Pro-TC-8 ISO

### Beispiel

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
    REM Set sampling rate to 27.5 Hz
    TC_SET_RATE(1, 8)
```

`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 ADwinPRO_ALL.inc
```

```
CAN_MSG[n] = value
```

or

```
ret_val = CAN_MSG[n]
```

## Parameters

<code>n</code>	Number of an array element (1... 9).	LONG
<code>value</code>	Expression whose value (0...256) is written into the message object.	LONG
<code>ret_val</code>	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 the instruction **TRANSMIT**.

## See also

[EN\\_RECEIVE](#), [EN\\_TRANSMIT](#), [READ\\_MSG](#), [TRANSMIT](#)

## To be used for the modules

CAN-1, CAN-2

## CAN\_MSG

### Example

REM Sends a 32 Bit FLOAT-value (here: Pi) as sequence of  
REM 4 bytes in a message object  
REM (Receiving of a float value see example at [READ\\_MSG](#))

```
#INCLUDE ADwinPRO_ALL.inc
#DEFINE pi 3.14159265
DIM i AS LONG

INIT:
    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)
    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:
    TRANSMIT(1,1,6)         'Send the message object 6
```



**EN\_INTERRUPT** configures a message object of the specified module to generate an external event (interrupt) when a message arrives.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

EN_INTERRUPT (module, channel, objectno)
```

## Parameters

<b>module</b>	Specified module address (1...255).	LONG
<b>channel</b>	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
<b>objectno</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 **EVENTENABLE**. 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.

## See also

[EN\\_RECEIVE](#), [EVENTENABLE](#), [GET\\_CAN\\_REG](#), [INIT\\_CAN](#)

## To be used for the modules

CAN-1, CAN-2

## Example

```
#INCLUDE ADwinPRO_ALL.inc
INIT:
    REM Initialize CAN controller 1 on the CAN module 1
    INIT_CAN(1,1)
    REM Configure message objects 3 and 15 for read
    EN_RECEIVE(1,1,3,1,0)
    EN_RECEIVE(1,1,15,385,0)
    REM Configure interrupt for message objects 3 and 15
    EN_INTERRUPT(1,1,3)
    EN_INTERRUPT(1,1,15)
    REM Enable event signal
    EVENTENABLE(1,ext,1)
EVENT:
    REM Read interrupt register (see below)
    PAR_13 = 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
```

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

## EN\_INTERRUPT



## EN\_RECEIVE

**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
EN_RECEIVE (module, channel, objectno, id, extend)
```

### Parameters

module	Specified module address (1...255).	LONG
channel	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
objectno	Number (1...15) of the message object in the CAN controller.	LONG
id	Identifier of the messages which are to be received in this message object (0...2 <sup>11</sup> or 0...2 <sup>29</sup> ).	LONG
extend	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 **EN\_RECEIVE**.

### See also

[CAN\\_MSG](#), [EN\\_TRANSMIT](#), [INIT\\_CAN](#), [READ\\_MSG](#), [TRANSMIT](#)

### To be used for the modules

CAN-1, CAN-2

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
REM Initialization of the CAN controller 1 on the CAN module 1
INIT_CAN (1,1)
REM Enable message object 1 to receive CAN messages
REM with the 11 bit-identifier 200
EN_RECEIVE (1,1,1,200,0)
```

**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
EN_TRANSMIT (module, channel, objectno, id, extend)
```

## Parameters

module	Specified module address (1...255).	LONG
channel	Number (1, 2) of the CAN channel that specifies the CAN controller.	LONG
objectno	Number (1...14) of the message object in the CAN controller.	LONG
id	Identifier of the messages that are sent in this message object (0...2 <sup>11</sup> oder 0...2 <sup>29</sup> ).	LONG
extend	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 **EN\_TRANSMIT**.

## See also

[CAN\\_MSG](#), [EN\\_RECEIVE](#), [INIT\\_CAN](#), [READ\\_MSG](#), [TRANSMIT](#)

## To be used for the modules

CAN-1, CAN-2

## Example

```
#INCLUDE ADwinPRO_ALL.inc
INIT:
    REM Initialization of the CAN controller 1 on the CAN module 1
    INIT_CAN(1,1)
    REM Enable message object 6 for sending of CAN messages
    REM with the 11 bit-identifier 40
    EN_TRANSMIT(1,1,6,40,0)
```

## EN\_TRANSMIT

## GET\_CAN\_REG

**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 = GET_CAN_REG(module, channel, regno)
```

### Parameters

module	Specified module address (1...255).	LONG
channel	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
regno	Register number of the CAN controller (0...255).	LONG
ret_val	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®), e.g.:

- address 00h: control register
- address 01h: status register
- address 5fh: interrupt register

### See also

[EN\\_INTERRUPT](#), [INIT\\_CAN](#), [SET\\_CAN\\_BAUDRATE](#), [SET\\_CAN\\_REG](#)

### To be used for the modules

CAN-1, CAN-2

### Example

```
#INCLUDE ADwinPRO_ALL.inc  
INIT:  
  REM Initialization of the CAN controller 1 on the CAN module 1  
  INIT_CAN(1,1)  
  REM Read out the control register of CAN controller 1, module 1  
  PAR_1 = GET_CAN_REG(1,1,0)
```

**INIT\_CAN** initializes one of the CAN controllers on the specified module and sets it into an initial status.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
INIT_CAN(module, channel)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>channel</code>	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 1 MBit/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 **SET\_CAN\_BAUDRATE**.

## See also

EN\_RECEIVE, EN\_TRANSMIT, GET\_CAN\_REG, SET\_CAN\_BAUDRATE, SET\_CAN\_REG

## To be used for the modules

CAN-1, CAN-2

## Example

```
#INCLUDE ADwinPRO_ALL.inc
INIT:
    REM Initialization of the CAN controller 1 on the CAN module 1
    INIT_CAN(1,1)
```

## INIT\_CAN



## READ\_MSG

**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 = READ_MSG(module, channel, msgno)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>channel</code>	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
<code>msgno</code>	Number (1... 15) of the message object in the CAN controller.	LONG
<code>ret_val</code>	≥-1: A new message has arrived, the value is the identifier of the message object. -1: No new message has arrived.	LONG

### Notes

The message object must be enabled before for receiving data with the instruction **EN\_TRANSMIT**.

If the message object receives a new message, its data are read out and copied to the array `CAN_MSG[]`. In this case the return value corresponds to the identifier of the message received. The bits indicating that a message has been received are reset so that a new message can be received.

The instruction to be used for querying the message objects in the polling mode, because here a check is made first, if new messages have arrived. If this is not the case no error occurs in the program.

### See also

[CAN\\_MSG](#), [EN\\_RECEIVE](#), [EN\\_TRANSMIT](#), [INIT\\_CAN](#), [TRANSMIT](#)

### To be used for the modules

CAN-1, CAN-2

## Example

REM If a new message with the correct identifier is received  
REM the data is read out. The first 4 bytes of the message are  
REM combined to a float value of length 32 bit. (Sending a  
REM float value see example of [TRANSMIT](#)).

```
#INCLUDE ADwinPRO_ALL.inc
```

```
DIM n AS LONG
```

```
INIT:
```

```
PAR_1 = 0
```

```
  INIT_CAN(1,1)           'Initialize CAN controller 1
```

```
  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
```

## SET\_CAN\_BAUDRATE

**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 = SET_CAN_BAUDRATE(module, channel, rate)
```

### Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>channel</code>	Number (1, 2) of the CAN channel, that determines the CAN controller.	LONG
<code>rate</code>	Baud rate of the CAN controller: High speed CAN: 5000...1 000 000 Bit/s. Low speed CAN: 5000 ... 125 000 Bit/s.	LONG
<code>ret_val</code>	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 **INIT\_CAN**, because otherwise the set Baud rate will be overwritten by the default setting (1 MBit/s).



### See also

[GET\\_CAN\\_REG](#), [INIT\\_CAN](#), [SET\\_CAN\\_REG](#)

### To be used for the modules

CAN-1, CAN-2

### Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM status AS LONG
INIT:
    INIT_CAN(1,1) 'Initialization of the CAN controller
    status = SET_CAN_BAUDRATE(1,1,125000) 'Set Baud rate 125 kBit/s
```



## 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	

## SET\_CAN\_REG

**SET\_CAN\_REG** writes a value in a register of the selected CAN controller on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
SET_CAN_REG(module, channel, regno, value)
```

### Parameters

module	Specified module address (1...255).	LONG
channel	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
regno	Register number (0...255) of the CAN controller.	LONG
value	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 control register has the address 0 and the status register the address 1.

### See also

[INIT\\_CAN](#), [SET\\_CAN\\_BAUDRATE](#), [GET\\_CAN\\_REG](#)

### To be used for the modules

CAN-1, CAN-2

### Example

```
#INCLUDE ADwinPRO_ALL.inc  
INIT:  
  INIT_CAN(1,1)           'Initialization of the CAN controller  
  SET_CAN_REG(1,1,0,1)    'Set control register to the value 1
```

**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

TRANSMIT (module, channel, msgno)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>channel</code>	Number (1, 2) of the CAN channel, that determines the CAN controller.	LONG
<code>msgno</code>	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 **EN\_TRANSMIT**.

The message data must be entered in `CAN_MSG []`, before executing the instruction **TRANSMIT**.

## See also

[CAN\\_MSG](#), [EN\\_RECEIVE](#), [EN\\_TRANSMIT](#), [READ\\_MSG](#)

## To be used for the modules

CAN-1, CAN-2

## Example

```
REM Sends a 32 Bit FLOAT-value (here: Pi) as sequence of
REM 4 bytes in a message object
REM (Receiving of a float value see example at READ_MSG)
```

```
#INCLUDE ADwinPRO_ALL.inc
#DEFINE pi 3.14159265
DIM i AS LONG

INIT:
  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)
  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:
  TRANSMIT(1,1,6)         'Send the message object 6
```

## TRANSMIT

## CHANGED\_DATA

**CHANGED\_DATA** checks, if the ndata in the output area have been changed since the user's last access to the DP-RAM

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = CHANGED_DATA(module, outsize)
```

### Parameters

module	Specified module address (1...4).	LONG
outsize	Size in bytes of the output area to check.	LONG
ret_val	Flag for data changes in the output area: 0 Data have not been changed. ≠0 New data available.	LONG

### Notes

This function is used to save computing time, by only reading out the data when they are changed.

The flag is reset when the right to access the output area changes from the application to the fieldbus. It doesn't matter if the function **CHANGED\_DATA** is called or not.

The function can only be used when you have released it during initialization (see [INIT\\_SLAVE](#)).

### To be used for the modules

Inter-SL, Profi-SL

### See also

[CHECK\\_ACCESS](#), [GET\\_PRO\\_BYTE](#), [GET\\_READ\\_BUFFER](#), [INIT\\_SLAVE](#), [REQUEST\\_ACCESS](#), [REQUEST\\_RELEASE\\_ACCESS](#), [SET\\_PRO\\_BYTE](#), [SET\\_WRITE\\_BUFFER](#)

### Example

```
#INCLUDE ADwinPRO_ALL.inc
```

#### EVENT:

```
REQUEST_ACCESS(1,2)      'Request access to the DP-RAM
                          '(Output area)
PAR_1 = CHECK_ACCESS(1)   'Get access rights status
IF (PAR_1 = 2) THEN       'If ADwin has access rights ...
    IF (CHANGED_DATA(1,10) <> 0) THEN 'and if there are new data ...
        PAR_2 = GET_PRO_BYTE(1,10) 'read a byte
    ENDIF
ENDIF
REQUEST_RELEASE_ACCESS(1,2) 'Release access to the DPM-RAM
```

See also programming example "[Data exchange with fieldbus](#)" on [page 330](#).



**CHECK\_ACCESS** returns to which areas of the DP-RAM the application has access rights.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = CHECK_ACCESS (module)
```

## Parameters

<b>module</b>	Specified module address (1...4).	LONG
<b>ret_val</b>	Bit pattern for access status of the areas: Bit = 0: Application has no access right. Bit = 1: Application has access right.	LONG

Bit no.	31:3	2	1	0
Data area	–	Input	Output	Control register

## Notes

If the application has no access right for a data area of the DP-RAM, you cannot access this area. This is necessary because otherwise the data will become inconsistent and the system may not work appropriately.

## To be used for the modules

Inter-SL, Profi-SL

## See also

CHANGED\_DATA, GET\_PRO\_BYTE, GET\_READ\_BUFFER, INIT\_SLAVE, REQUEST\_ACCESS, REQUEST\_RELEASE\_ACCESS, SET\_PRO\_BYTE, SET\_WRITE\_BUFFER

## Example

```
#INCLUDE ADwinPRO_ALL.inc
```

### EVENT:

```
REQUEST_ACCESS (1,1)      'Request access to the DP-RAM
                           '(Control register).
PAR_1 = CHECK_ACCESS (1)  'Check, if ADwin has access rights
IF (PAR_1 = 1) THEN      'If there is access right, ...
    PAR_2 = GET_PRO_BYTE (1,7F6h) 'read a byte
ENDIF
REQUEST_RELEASE_ACCESS (1,1) 'Release access to the DP-RAM.
```

See also programming example "Data exchange with fieldbus" on [page 330](#).

## CHECK\_ACCESS

## GET\_PRO\_BYTE

**GET\_PRO\_BYTE** returns a byte of a specified memory address of the DP-RAM of the fieldbus module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
ret_val = GET_PRO_BYTE(module,byteno)
```

### Parameter

module	Specified module address (1...4).	LONG
byteno	Memory address in the DP-RAM.	LONG
ret_val	Contents of the memory address of the DP-RAM (0...256).	LONG

### Notes

The function accesses each memory location, regardless whether the application has access right or not. If the application reads out data without access right, incorrect data may be transferred or a bus error may occur.

Therefore pay attention to not using the function in an unauthorized area or period of time.

### To be used for the modules

Inter-SL, Profi-SL

### See also

CHANGED\_DATA, CHECK\_ACCESS, GET\_READ\_BUFFER, INIT\_SLAVE, REQUEST\_ACCESS, REQUEST\_RELEASE\_ACCESS, SET\_PRO\_BYTE, SET\_WRITE\_BUFFER

### Example

```
#INCLUDE ADwinPRO_ALL.inc  
  
INIT:  
  PAR_1 = GET_PRO_BYTE(1,7CDh) 'The contents of byte number 7CDh  
                                '(fieldbus type) is assigned to PAR_1
```





**GET\_READ\_BUFFER** copies a defined data block from the memory area of the DP-RAM into the specified destination array.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
GET_READ_BUFFER(module,array[],start,count)
```

## Parameter

module	Specified module address (1...4).	LONG
array[]	Name of the destination array.	ARRAY LONG
start	First element in the DP-RAM of the data block, which is to copy.	LONG
count	Amount of data bytes to copy from the DP-RAM.	LONG

## Notes

You may only use this instruction when the application has the access right for the DP-RAM.

The destination array into which the data are transferred, must already be declared, at least with so many array elements as data bytes are copied.

## To be used for the modules

Inter-SL, Profi-SL

## See also

CHANGED\_DATA, CHECK\_ACCESS, GET\_PRO\_BYTE, INIT\_SLAVE, REQUEST\_ACCESS, REQUEST\_RELEASE\_ACCESS, SET\_PRO\_BYTE, SET\_WRITE\_BUFFER

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM DATA_1[100] AS LONG

EVENT:
  REQUEST_ACCESS(1,2)      'Request access to the DP-RAM
                           '(output area)
  PAR_1 = CHECK_ACCESS(1)  'Read access right status
  IF (PAR_1 = 2) THEN      'If ADwin has access right...
    IF (CHANGED_DATA(1,10) <> 0) THEN 'and new data are available
      GET_READ_BUFFER(1,DATA_1,0,10) 'read 10 bytes
    ENDIF
  ENDIF
  REQUEST_RELEASE_ACCESS(1,2) 'Release access to the DP-RAM
```

See also programming example "Data exchange with fieldbus" on [page 330](#).

## GET\_READ\_BUFFER

## INIT\_SLAVE

**INIT\_SLAVE** initializes the fieldbus slave and can only be used after power up.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = INIT_SLAVE(module, IO_in, par_in, IO_out,
par_out, in_hdl, out_hdl, intr)
```

### Parameters

module	Specified module address (1...4).	LONG
IO_in	Size of the input data area in bytes for cyclic data.	LONG
par_in	Length of the input data area in bytes for acyclic data.	LONG
IO_out	Length of the output data area in bytes for cyclic data.	LONG
par_out	Length of the output data area in bytes for acyclic data.	LONG
in_hdl	Bit pattern for the handling of the input data area: .	LONG

Bit no.	31:2	1	0
Bit = 0	–	Disable flag for <b>CHANGED_DATA</b> .	Clear input area as soon as the application stops.
Bit = 1	–	Enable flag for <b>CHANGED_DATA</b> .	Freeze input area as soon as the application stops.

out_hdl	Bit pattern for the handling of the output data area: .	LONG
---------	---	------

Bit no.	31:2	1	0
Bit = 0	–	Clear output area as soon as the fieldbus stops.	Start fieldbus off-line.
Bit = 1	–	Freeze output area as soon as the fieldbus stops.	Start fieldbus on-line.

intr	Bit pattern for the handling of the interrupt.	LONG
------	--	------

Bit no.	31:2	1	0
	–	Release interrupt when the fieldbus is on-line:	Release interrupt when the flag for <b>CHANGED_DATA</b> is enabled:
Bit = 0.	–	No	No
Bit = 1.	–	Yes	Yes

`ret_val` Bit pattern as status of the initialization: LONG  
 Bit = 0: no error.  
 Bit = 1: error occurred (meaning see below).

Bit-Nr.	31:16	15	14	13:8	7	6	5:3	2	1	0
Fehler.	–	A <sub>7</sub>	A <sub>6</sub>	–	A <sub>5</sub>	A <sub>4</sub>	–	A <sub>3</sub>	A <sub>2</sub>	A <sub>1</sub>

A<sub>1</sub>: Error during hardware check.  
 A<sub>2</sub>: Error during start of the initialization.  
 A<sub>3</sub>: Error during the initialization.  
 A<sub>4</sub>: Error in the DP-RAM.  
 A<sub>5</sub>: Error at the end of initialization.  
 A<sub>6</sub>: No message received.  
 A<sub>7</sub>: No message sent.

## Notes

This instruction must be executed before you start working with the slave module.

A second initialization after power up is not possible.

During the initialization with **INIT\_SLAVE**, the size of the input and output areas is set (individually for cyclic and acyclic data). The size has to match with the specified size in the corresponding master. The maximum size of the individual areas differs according to the fieldbus type.

After a successful initialization the slave is ready to exchange data and parameters can be read out from the DP-RAM. The application is only allowed to write information into the DP-RAM when it has access rights!

If incorrect data have been exchanged during the first initialization, the system must be turned off. Only after a new power up can the module be reinitialized.

With this instruction a sequence is processed, which takes approx. 2-3 seconds. If the instruction is called in a high-priority process (that cannot be interrupted), there will be no communication between computer and ADwin system. Therefore we recommend to initialize in a low-priority process or in a process section with low priority (e.g. **LOWINIT**:) .



## To be used for the modules

Inter-SL, Profi-SL

## See also

**CHANGED\_DATA**, **CHECK\_ACCESS**, **GET\_PRO\_BYTE**, **GET\_READ\_BUFFER**, **REQUEST\_ACCESS**, **REQUEST\_RELEASE\_ACCESS**, **SET\_PRO\_BYTE**, **SET\_WRITE\_BUFFER**

## Example

```
#INCLUDE ADwinPRO_ALL.inc
```

### INIT:

```
REM initialize slave:
REM Only cyclic data (10 IO_in / 10 IO_out),
REM CHANGE_DATA function ON, no interrupts,
REM Outputs are cleared, when the bus is OFF-line.
PAR_1 = INIT_SLAVE(1,10,0,10,0,2,0,0)
```

## REQUEST\_ACCESS

**REQUEST\_ACCESS** requests access to the DP-RAM of the slave.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
REQUEST_ACCESS (module, area)
```

### Parameters

module	Specified module address (1...4).	LONG
area	Bit pattern for the area whose access status is changed: Bit = 0: Access right remains unchanged. Bit = 1: Access right is requested.	LONG

Bit no.	31:3	2	1	0
Data area	–	Input	Output	Control register

### Notes

If access rights are requested for several areas at the same time, it may happen that the fieldbus side refuses access to a partial area. The other areas can nevertheless be accessed.

If the application has no access right to the DP-RAM, the area must not be accessed, because data may become incorrect and the system unstable.

After data exchange with the DP-RAM the access right should be returned to the fieldbus side again with **REQUEST\_RELEASE\_ACCESS**, so that the data can be transferred to the master and the data can be written to the DP-RAM. If the application does not return the access right to the bus, it will be automatically done after 1 second.

### To be used for the modules

Inter-SL, Profi-SL

### See also

[CHANGED\\_DATA](#), [CHECK\\_ACCESS](#), [GET\\_PRO\\_BYTE](#), [GET\\_READ\\_BUFFER](#), [INIT\\_SLAVE](#), [REQUEST\\_RELEASE\\_ACCESS](#), [SET\\_PRO\\_BYTE](#), [SET\\_WRITE\\_BUFFER](#)

### Example

```
#INCLUDE ADwinPRO_ALL.inc  
  
EVENT:  
  REQUEST_ACCESS (1,1)           'Request access to the DP-RAM  
                                   '(Control register)  
  PAR_1 = CHECK_ACCESS (1)       'Get access rights status  
  IF (PAR_1 = 1) THEN             'If ADwin has access right...  
    PAR_2 = GET_PRO_BYTE (1,7F6h) 'read a byte  
  ENDIF  
  REQUEST_RELEASE_ACCESS (1,1) 'Return access to the DP-RAM
```

See also programming example "[Data exchange with fieldbus](#)" on [page 330](#).



**REQUEST\_RELEASE\_ACCESS** requests to return the access right for the DP-RAM of the slave to the fieldbus.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

REQUEST_RELEASE_ACCESS (module, area)
```

## Parameters

<b>module</b>	Specified module address (1...4).	LONG
<b>area</b>	Bit pattern for the area, whose access status is changed: Bit = 0: Access right remains unchanged. Bit = 1: Access right is returned.	LONG

Bit no.	31:3	2	1	0
Data area	–	Input	Output	Control register

## Notes

The access right can be returned for several areas at the same time.

After data exchange with the DP-RAM the access right should be returned to the fieldbus side again, so that the data can be transferred to the master and the data can be written to the DP-RAM. If the application does not return the access right to the bus, it will be automatically done after 1 second.

## To be used for the modules

Inter-SL, Profi-SL

## See also

CHANGED\_DATA, CHECK\_ACCESS, GET\_PRO\_BYTE, GET\_READ\_BUFFER, INIT\_SLAVE, REQUEST\_ACCESS, SET\_PRO\_BYTE, SET\_WRITE\_BUFFER

## Example

```
#INCLUDE ADwinPRO_ALL.inc

EVENT:
REQUEST_ACCESS (1,1)      'Request access to the DP-RAM
                           '(Control register).
PAR_1 = CHECK_ACCESS (1)  'Get access rights status
IF (PAR_1 = 1) THEN      'If ADwin has access right...
    PAR_2 = GET_PRO_BYTE (1,7F6h) 'a byte is read.
ENDIF
REQUEST_RELEASE_ACCESS (1,1) 'Release access to the DP-RAM.
```

See also programming example "Data exchange with fieldbus" on page 330.

## REQUEST\_RELEASE\_ACCESS

## SET\_PRO\_BYTE

**SET\_PRO\_BYTE** sets a byte in the DP-RAM of the fieldbus slave.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
SET_PRO_BYTE(module,byteno,value)
```

### Parameters

module	Specified module address (1...4).	LONG
byteno	Memory address in the DP-RAM.	LONG
value	Value to be written into the memory address (0...255).	LONG

### Notes

The function accesses every byte, regardless whether the application has access rights or not. If the application reads out data without having access rights, incorrect data may be transferred or a bus error may occur.



Therefore pay attention to not using the function in an unauthorized area or period of time.

### To be used for the modules

Inter-SL, Profi-SL

### See also

[CHANGED\\_DATA](#), [CHECK\\_ACCESS](#), [GET\\_PRO\\_BYTE](#), [GET\\_READ\\_BUFFER](#), [INIT\\_SLAVE](#), [REQUEST\\_ACCESS](#), [REQUEST\\_RELEASE\\_ACCESS](#), [SET\\_WRITE\\_BUFFER](#)

### Example

```
#INCLUDE ADwinPRO_ALL.inc  
INIT:  
    REM Set byte number 0h to the value 2 (cyclic input data)  
    SET_PRO_BYTE(1,0h,2)
```

**SET\_WRITE\_BUFFER** copies the data from an array into a specified memory area of the DP-RAM.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
SET_WRITE_BUFFER(module,array[],start,count)
```

## Parameters

module	Specified module address (1...4).	LONG
array[]	Name of the source array.	ARRAY LONG
start	First memory byte in the DP-RAM into which data are written .	LONG
count	Amount of memory bytes in the DP-RAM into which data are written.	LONG

## Notes

The instruction can only be used, when the application has the access rights for the DP-RAM.

The source array that provides the data, must already be declared, at least with so many array elements as data bytes are copied.

## To be used for the modules

Inter-SL, Profi-SL

## See also

[CHANGED\\_DATA](#), [CHECK\\_ACCESS](#), [GET\\_PRO\\_BYTE](#), [GET\\_READ\\_BUFFER](#), [INIT\\_SLAVE](#), [REQUEST\\_ACCESS](#), [REQUEST\\_RELEASE\\_ACCESS](#), [SET\\_PRO\\_BYTE](#)

## Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[100] AS LONG

EVENT:
  REQUEST_ACCESS(1,4)      'request access to the DP-RAM
                           '(input area)
  PAR_1 = CHECK_ACCESS(1)  'Get access rights status
  IF (PAR_1 = 2) THEN      'If ADwin has access right...
    IF (CHANGED_DATA(1,10) <> 0) THEN 'and if there are new data
      SET_WRITE_BUFFER(1,DATA_1,0,10) 'transfer 10 bytes
    ENDIF
  ENDIF
  REQUEST_RELEASE_ACCESS(1,4) 'Release access to the DP-RAM
```

See also programming example "[Data exchange with fieldbus](#)" on [page 330](#).

## SET\_WRITE\_BUFFER

## CHECK\_SHIFT\_REG

**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 = CHECK_SHIFT_REG(module, register)
```

### Parameters

module	Specified module address (1...4).	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

[CHECK\\_SHIFT\\_REG](#), [READ\\_FIFO](#), [RS\\_INIT](#), [RS\\_RESET](#), [RS485\\_SEND](#), [SET\\_RS](#), [WRITE\\_FIFO](#)

### To be used for the modules

RS232-2, RS232-4, RS422-2, RS422-4, RS485-2, RS485-4

### Example

```
#INCLUDE ADwinPRO_ALL.inc  
  
EVENT:  
...  
PAR_1 = CHECK_SHIFT_REG(1,1) 'Check if channel 1 still  
                               'has data to send  
...
```



**GET\_RS** reads out the controller register on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = GET_RS(module, register)
```

## Parameters

module	Specified module address (1...4).	LONG
register	Address of the controller register to read.	LONG
ret_val	Contents of the controller register.	LONG

## Notes

We recommend to use this instruction only after you have more experience about how the controller operates (data-sheet of the manufacturer Texas Instruments). For more common applications more comfortable instructions are available in the include file.

## See also

[CHECK\\_SHIFT\\_REG](#), [READ\\_FIFO](#), [RS\\_INIT](#), [RS\\_RESET](#), [RS485\\_SEND](#), [SET\\_RS](#), [WRITE\\_FIFO](#)

## To be used for the modules

RS232-2, RS232-4, RS422-2, RS422-4, RS485-2, RS485-4

## Example

-/-

## GET\_RS

## READ\_FIFO

**READ\_FIFO** reads a value from the input FIFO of a specified channel on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = READ_FIFO(module, channel)
```

### Parameters

module	Specified module address (1...4).	LONG
channel	number of the channel that is to be read out (1, 2 or 1...4).	LONG
ret_val	Contents of the input FIFO: -1: FIFO is empty. ≥0: Transferred value.	LONG

### Notes

-/-

### See also

[CHECK\\_SHIFT\\_REG](#), [GET\\_RS](#), [RS\\_INIT](#), [RS\\_RESET](#), [RS485\\_SEND](#), [SET\\_RS](#), [WRITE\\_FIFO](#)

### To be used for the modules

RS232-2, RS232-4, RS422-2, RS422-4, RS485-2, RS485-4

### Example

```
#INCLUDE ADwinPRO_ALL.inc

INIT:
  RS_RESET(1)
  RS_INIT(1,1,9600,0,8,0,1) 'Initialization of channel 1 on module
                             '1 with 9600 Baud, without parity,
                             '8 data bits, 1 stop bit and
                             'hardware handshake (RS232 only).

EVENT:
  PAR_1 = READ_FIFO(1,1)    'Get a value from the FIFO. If
                             'the FIFO is empty, -1 is returned.
```

See also further [Examples for RS232 and RS485](#) from [page 331](#).

**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

RS_INIT(module, channel, baud, parity, bits, stop,
        handshake)
```

## Parameters

<code>module</code>	Specified module address (1...4).	LONG
<code>channel</code>	Channel, which is to be initialized (1, 2 or 1...4).	LONG
<code>baud</code>	Transfer rate in Baud: RS232: 35 ... 115200 Baud. RS485: 35...2304000 Baud.	LONG
<code>parity</code>	Use of test bits: 0: without parity bit. 1: even parity. 2: odd parity.	LONG
<code>bits</code>	Amount of data bits (5, 6, 7 or 8).	LONG
<code>stop</code>	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
<code>handshake</code>	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 the instruction **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	

## RS\_INIT



**See also**

CHECK\_SHIFT\_REG, GET\_RS, READ\_FIFO, RS\_RESET, RS485\_SEND, SET\_RS, WRITE\_FIFO

**To be used for the modules**

RS232-2, RS232-4, RS422-2, RS422-4, RS485-2, RS485-4

**Example**

```
#INCLUDE ADwinPRO_ALL.inc
```

**INIT:**

```
RS_RESET(1)                'Reset RS-module
RS_INIT(1,1,9600,0,8,0,1)    'Initialization of channel 1 to
                             'module 1 with 9600 Baud, without,
                             'parity, 8 data bits, 1 stop bit and
                             'hardware handshake (RS232 only).
```

**RS\_RESET** executes a hardware reset on the specified module and deletes the settings for all channels.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
RS_RESET(module)
```

## Parameters

**module** Specified module address (1...4). 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 **RS\_INIT** must follow, in order to initialize the controller and to set the interface parameters.

The instruction **RS\_INIT** sets the same registers as a hardware reset does. Nevertheless, **RS\_RESET** should be used for the case the controller has crashed.

## See also

[CHECK\\_SHIFT\\_REG](#), [GET\\_RS](#), [READ\\_FIFO](#), [RS\\_INIT](#), [RS485\\_SEND](#), [SET\\_RS](#), [WRITE\\_FIFO](#)

## To be used for the modules

RS232-2, RS232-4, RS422-2, RS422-4, RS485-2, RS485-4

## Example

```
#INCLUDE ADwinPRO_ALL.inc
```

```
INIT:
  RS_RESET(1)           'Reset RS-module
  RS_INIT(1,1,9600,0,8,0,1) 'Initialization of channel 1 to
                           'module 1 with 9600 Baud, without
                           'parity, 8 data bits, 1 stop bit and
                           'hardware handshake (RS232 only).
```

See also further [Examples for RS232 and RS485](#) from [page 331](#).

## RS\_RESET

## RS485\_SEND

**RS485\_SEND** determines the transfer direction for a specified channel on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc  
  
RS485_SEND (module, channel, dir)
```

### Parameters

module	Specified module address (1...4).	LONG
channel	Channel to be set (1, 2 or 1...4).	LONG
dir	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

[CHECK\\_SHIFT\\_REG](#), [GET\\_RS](#), [READ\\_FIFO](#), [RS\\_INIT](#), [RS\\_RESET](#), [SET\\_RS](#), [WRITE\\_FIFO](#)

### To be used for the modules

RS485-2, RS485-4

### Example

See also further [Examples for RS232 and RS485](#) from [page 331](#).

**SET\_RS** writes a value into a specified register on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
SET_RS(module, register, value)
```

## Parameters

<code>module</code>	Specified module address (1...4).	LONG
<code>register</code>	Number of the register, into which data are written.	LONG
<code>value</code>	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

[CHECK\\_SHIFT\\_REG](#), [GET\\_RS](#), [READ\\_FIFO](#), [RS\\_INIT](#), [RS\\_RESET](#), [RS485\\_SEND](#), [WRITE\\_FIFO](#)

## To be used for the modules

RS232-2, RS232-4, RS422-2, RS422-4, RS485-2, RS485-4

## Example

-/-

## SET\_RS

## WRITE\_FIFO

**WRITE\_FIFO** writes a value into the send-FIFO of a specified channel on the specified module.

### Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = WRITE_FIFO(module, channel, value)
```

### Parameters

module	Specified module address (1...4).	LONG
channel	Channel number to whose send-FIFO data are transferred (1, 2 or 1...4).	LONG
value	Value to be written into the send-FIFO.	LONG
ret_val	Status message: 0: Data are transferred successfully. 1: Data were not transferred, send-FIFO is full.	LONG

### Notes

The instruction checks first if there is at least one 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.

### See also

[CHECK\\_SHIFT\\_REG](#), [GET\\_RS](#), [READ\\_FIFO](#), [RS\\_INIT](#), [RS\\_RESET](#), [RS485\\_SEND](#), [SET\\_RS](#)

### Can be used for the modules

RS232-2, RS232-4  
RS485-2, RS485-4

### Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM val AS LONG

INIT:
  RS_RESET(1)
  RS_INIT(1,1,9600,0,8,0,1) 'Initialization of channel 1 to
                             '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).
```

See also further [Examples for RS232 and RS485](#) from [page 331](#).



**LS\_DIO\_INIT** initializes the specified module of type HSM-24V on the LS bus via an interface of the Pro module and returns the error status

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = LS_DIO_INIT(module, channel, ls-module)
```

## Parameters

module	Specified module address (1...255).	LONG
channel	number (1, 2) of the LS bus interface on the Pro module.	LONG
ls-module	Specified module address on the LS bus (1...15).	LONG
ret_val	Bit pattern representing the error status. Bit = 0: No error. Bit = 1: Error occurred.	LONG

Bit no.	31...8	7	6	5...4	3	2	1	0
Status	–	Temp2	Temp1	–	WD	Time	Ovr	Par

- :don't care (mit 0CFh ausmaskieren)

Par: Parity-Fehler bei der Datenübertragung auf dem LS-Bus.

Ovr: Overrun-Fehler bei der Datenübertragung auf dem LS-Bus.

Time: Timeout-Fehler bei der Datenübertragung auf dem LS-Bus.

WD: Watchdog hat ausgelöst. Die Kanaltreiber sind deaktiviert.

Temp1: Übertemperatur am Treiber für Kanäle 1...16. Treiber ist deaktiviert.

Temp2: Übertemperatur am Treiber für Kanäle 17...32. Treiber ist deaktiviert.

## Notes

The instruction only be used in section **INIT** : , since it takes long processing time.

The initialization does the following settings:

- All DIO channels are set as inputs.  
Other settings see **LS\_DIGPROG**.
- The over-current status (> ca. 500mA) is reset.
- The error status for superheating is reset.
- The error status for timeout on the LS bus is reset.

The error "superheating" of a driver may only occur, if over-current in the range of 150...500mA is present on several channels at the same time. Irrespective of this, an over-current of mor than 500mA automatically switches off the concerned channel.

The channels of the module HSM-24V may only be operated in the range of 0...150mA.



## To be used for the modules

LS-2 Rev. A

## See also

[LS\\_DIGPROG](#), [LS\\_WATCHDOG\\_INIT](#), [LS\\_DIG\\_IO](#)

## Example

```
REM Example prozess for one module HSM-24V and ADwin-Pro-LS2
#include ADwinPRO_ALL.inc

INIT:
    PROCESSDELAY = 4000000    '10Hz HP
    PAR_1 = LS_DIO_INIT(1,2,1)
    PAR_2 = LS_DIGPROG(1,2,1,0Fh) 'channels 1...32 as output
    PAR_3 = LS_WATCHDOG_INIT(1,2,1,1,1100) 'watchdog time 1.1 sec

EVENT:
    REM set one channel to high, rotating from 1 to 32
    INC PAR_10
    IF (PAR_10>=32) THEN PAR_10=0
    PAR_11 = SHIFT_LEFT(1,PAR_10)
    REM set channels and read back real state
    PAR_12 = LS_DIG_IO(1,2,PAR_11)
```

**LS\_DIGPROG** sets the digital channels 1...32 of the specified module of type HSM-24V on the LS bus as inputs or outputs in groups of 8 via an interface of the Pro module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = LS_DIGPROG(module, channel, ls-module,
                    pattern)
```

## Parameters

**module** Specified module address (1...255). LONG

**channel** number (1, 2) of the LS bus interface on the Pro module. LONG

**ls-module** Specified module address on the LS bus (1...15). LONG

**pattern** Bit pattern, setting the channels as inputs or outputs: LONG  
 Bit = 0: Set channels as inputs.  
 Bit = 1: Set channels as outputs.

Bit No.	31...4	3	2	1	0
Channel no.	–	32:25	24:17	16:9	8:1

**ret\_val** Bit pattern representing the error status. LONG  
 Bit = 0: No error.  
 Bit = 1: Error occurred.

Bit no.	31...8	7	6	5...4	3	2	1	0
Status	–	Temp2	Temp1	–	WD	Time	Ovr	Par

- :don't care (mask with 0CFh).

Par:Parity error during data transfer on the LS bus.

Ovr:Overflow error during data transfer on the LS bus.

Time:Timeout error during data transfer on the LS bus.

WD:Watchdog was released. The channel drivers are deactivated.

Temp1:Superheating on driver for channels 1...16. Driver is deactivated.

Temp2:Superheating on driver for channels 17...32. Driver is deactivated.

## Notes

The instruction only be used in section **INIT** , since it takes long processing time.

After initialization with **LS\_DIO\_INIT** all channels are set as inputs.

The channels may be set as inputs or outputs in groups of 8 only (4 relevant bits only, other bits are ignored).

## To be used for the modules

LS-2 Rev. A

## See also

[LS\\_DIO\\_INIT](#), [LS\\_WATCHDOG\\_INIT](#), [LS\\_DIG\\_IO](#)

## LS\_DIGPROG

### Example

```
REM Example prozess for one module HSM-24V and ADwin-Pro-LS2
#include ADwinPRO_ALL.inc

INIT:
    PROCESSDELAY = 4000000    '10Hz HP
    PAR_1 = LS_DIO_INIT(1,2,1)
    PAR_2 = LS_DIGPROG(1,2,1,0Fh) 'channels 1...32 as output
    PAR_3 = LS_WATCHDOG_INIT(1,2,1,1,1100) 'watchdog time 1.1 sec

EVENT:
    REM set one channel to high, rotating from 1 to 32
    INC PAR_10
    IF (PAR_10>=32) THEN PAR_10=0
    PAR_11 = SHIFT_LEFT(1,PAR_10)
    REM set channels and read back real state
    PAR_12 = LS_DIG_IO(1,2,PAR_11)
```

**LS\_DIG\_IO** sets all digital outputs of the specified module HSM-24V on the LS bus to the level High oder Low and returns the status of all channels as bit pattern.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc

ret_val = LS_DIG_IO(module, channel, pattern)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>channel</code>	number (1, 2) of the LS bus interface on the Pro module.	LONG
<code>pattern</code>	Bit pattern, setting the digital outputs (see table). Bit = 0: Set outputs to level Low. Bit = 1: Set outputs to level High.	LONG
<code>ret_val</code>	Bit pattern representing the real state of all digital channels (see table). Bit = 0: Channel has level Low. Bit = 1: Channel has level High.	LONG

Bit No.	31	30	29	...	2	1	0
Input no.	32	31	30	...	3	2	1

## Notes

**LS\_DIG\_IO** only runs correctly, if the following conditions are given:

- There is only one module on the LS bus.
- The module is of type HSM-24V.
- The module's address is set to 1.

The channels are set as inputs or outputs using **LS\_DIGPROG**.

The `pattern` is applied to those channels only, which are set as outputs. Bits for input channels are ignored.

The return value contains the real state of both inputs and outputs. The inputs have a filter causing about 12µs signal delay.

**LS\_DIG\_IO** resets the watchdog counter of the module to the start value. The counter remains enabled. The start value is set using **LS\_WATCHDOG\_INIT**.

Reset the active watchdog timer at least once to the start value within the counting interval, in order to keep the module working.

## To be used for the modules

LS-2 Rev. A

## See also

[LS\\_DIO\\_INIT](#), [LS\\_DIGPROG](#), [LS\\_WATCHDOG\\_INIT](#),

## LS\_DIG\_IO



## Example

```
REM Example prozess for one module HSM-24V and ADwin-Pro-LS2
#include ADwinPRO_ALL.inc

INIT:
    PROCESSDELAY = 4000000    '10Hz HP
    PAR_1 = LS_DIO_INIT(1,2,1)
    PAR_2 = LS_DIGPROG(1,2,1,0Fh) 'channels 1...32 as output
    PAR_3 = LS_WATCHDOG_INIT(1,2,1,1,1100) 'watchdog time 1.1 sec

EVENT:
    REM set one channel to high, rotating from 1 to 32
    INC PAR_10
    IF (PAR_10>=32) THEN PAR_10=0
    PAR_11 = SHIFT_LEFT(1,PAR_10)
    REM set channels and read back real state
    PAR_12 = LS_DIG_IO(1,2,PAR_11)
```

**LS\_WATCHDOG\_INIT** enables or disables the watchdog counter of a specified module on the LS bus via an interface of the Pro module.

If enabled, the counter is set to the start value and is started.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
ret_val = LS_WATCHDOG_INIT(module, channel,
    ls-module, enable, time)
```

## Parameters

<code>module</code>	Specified module address (1...255).	LONG
<code>channel</code>	number (1, 2) of the LS bus interface on the Pro module.	LONG
<code>ls-module</code>	Specified module address on the LS bus (1...15).	LONG
<code>enable</code>	Set status of watchdog counter: 0 : Disable watchdog counter. 1 : Enable watchdog counter.	LONG
<code>time</code>	Release time (0...107374) of the counter in milliseconds.	LONG
<code>ret_val</code>	Bit pattern representing the error status. Bit = 0: No error. Bit = 1: Error occurred.	LONG

Bit no.	31...8	7	6	5...4	3	2	1	0
Status	–	Temp2	Temp1	–	WD	Time	Ovr	Par
- : don't care (mask with 0CFh)								
Par: Parity error during data transfer on the LS bus.								
Ovr: Overrun error during data transfer on the LS bus.								
Time: Timeout error during data transfer on the LS bus.								
WD: Watchdog was released. The channel drivers are deactivated.								
Temp1: Superheating on driver for channels 1...16. Driver is deactivated.								
Temp2: Superheating on driver for channels 17...32. Driver is deactivated.								

## Notes

The instruction only be used in section **INIT** : , since it takes long processing time.

As long as the watchdog counter is enabled, it decrements the counter value continuously. After the set release time the counter value reaches 0 (zero). If so, the module assumes a malfunction and stops; thus, all output signals are reset.

After power-up of the module the counter is set to the start value 10ms and the watchdog counter is enabled.

Reset the active watchdog timer at least once to the start value within the counting interval, in order to keep the module working. To reset the module use any module specific instruction or **LS\_WATCHDOG\_RESET**.

The watchdog function is used as to monitor the connection between ADwin system and LS bus module.

## LS\_WATCHDOG\_INIT



## To be used for the modules

LS-2 Rev. A

## See also

[LS\\_DIO\\_INIT](#), [LS\\_DIGPROG](#), [LS\\_DIG\\_IO](#)

## Example

REM Example prozess for one module HSM-24V and ADwin-Pro-LS2

**#INCLUDE** ADwinPRO\_ALL.inc

### INIT:

**PROCESSDELAY** = 4000000    '10Hz HP

**PAR\_1** = **LS\_DIO\_INIT**(1,2,1)

**PAR\_2** = **LS\_DIGPROG**(1,2,1,0Fh) 'channels 1...32 as output

**PAR\_3** = **LS\_WATCHDOG\_INIT**(1,2,1,1,1100) 'watchdog time 1.1 sec

### EVENT:

REM set one channel to high, rotating from 1 to 32

**INC** **PAR\_10**

**IF** (**PAR\_10**>=32) **THEN** **PAR\_10**=0

**PAR\_11** = **SHIFT\_LEFT**(1,**PAR\_10**)

REM set channels and read back real state

**PAR\_12** = **LS\_DIG\_IO**(1,2,**PAR\_11**)



### 4 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.

#### 4.1 Pro II: All Modules

All Pro II modules, which are accessed by active *ADbasic* programs, 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.



## P2\_CHECK\_LED

**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

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

### See also

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

### To be used for the modules

all modules for Pro II bus.

### 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
```

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.
<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.

## Notes

The instruction **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.

The instruction **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)

## To be used for the modules

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:
  ...
  IF (CPU_DIGIN(1) = 1) THEN 'If falling edge has been detected ...
    END                      '... end this program
  ENDIF
  ...
```

## CPU\_DIGIN

## CPU\_DIGOUT

**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

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

### Notes

The instruction **CPU\_DIGOUT** The instruction **CPU\_DIGIN** 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](#)

### To be used for the modules

CPU-T11

### Example

```
#INCLUDE ADwinPRO_ALL.INC  
  
EVENT:  
...  
CPU_DIGOUT(1,0)           'Set DIG I/O 1 to TTL level low  
...
```

**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 LONG and edge 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>		31:04	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](#)

## To be used for the modules

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)
...
```

## CPU\_DIG\_IO\_CONFIG

## CPU\_EVENT\_CONFIG

**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

min_hold	Minimum time, which an edge must be held to be accepted: 0: 15ns (Default). 1: 50ns.	LONG
edge	Type of edge which is accepted: 1: rising edge (Default). 2: falling edge. 3: rising and falling edge.	LONG
prescale	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 to the module Pro-OPT-16 and enable the module's external event input with **EVENTENABLE**.

### See also

[P2\\_EVENT\\_READ](#), [P2\\_EVENT\\_ENABLE](#), [EVENTENABLE](#)

### To be used for the modules

CPU-T11

### Example

```
#INCLUDE ADwinPRO_ALL.INC  
  
INIT:  
REM Configure input EVENT IN for minimum 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.  
...
```

**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, value)
```

## Parameters

<b>module</b>	Specified module address (1...15).	LONG
<b>value</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](#)

## To be used for the modules

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

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

### Notes

An event input must be enabled with the instruction **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.

### See also

[P2\\_EVENT\\_ENABLE](#), [P2\\_EVENT2\\_CONFIG](#), [P2\\_EVENT\\_READ](#)

### To be used for the modules

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

### Example

```
#INCLUDE ADwinPRO_ALL.INC

INIT:
REM Configure event input for mimimum 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. 2: Signal from AB mode. 3: Signal from AB mode after clearance.	LONG
<b>edge</b>	Type of edge, that is accepted at event inputs: 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\_EVENT\_CONFIG** and enabled with **P2\_EVENT\_ENABLE**.

According to the module different modes are available:

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

The module passes the signal at input **EVENT1** as resulting event signal. The parameter **edge** is of no importance.

The input **EVENT1** is disabled first, the resulting event signal is TTL level low. As soon as an edge of type **edge** arrives at **EVENT3**, the module enables input **EVENT1** and passes its signal as resulting event signal.

The input **EVENT1** is disabled again by setting mode 0.

In AB mode, the module evaluates two rectangular signals at the inputs **EVENT1** and **EVENT2**, which are phase-shifted by 90 degrees (typical for incremental encoders): If an edge of type **edge** arrives at one of the inputs, the the TTL level of the resulting event signal toggles.

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 **EVENT1** and an edge at **EVENT2** 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 betweenn 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 `EVENT1` and `EVENT2` are disabled first, the resulting event signal is TTL level low. As soon as an edge of type `edge` arrives at `EVENT3`, the module enables the inputs `EVENT1` and `EVENT2` and evaluates the two rectangular signals to the resulting event signal (see [Signal from AB mode](#)).

The inputs `EVENT1` and `EVENT2` are disabled again by setting mode 0.

**See also**

[P2\\_EVENT\\_ENABLE](#), [P2\\_EVENT\\_CONFIG](#), [P2\\_EVENT\\_READ](#)

**To be used for the modules**

Aln-F-8/14 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_EVENT2_CONFIG(1,0,2,4)  
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

<b>module</b>	Module address (0...15): 0: CPU module. 1...15: specified module address.	LONG
<b>ret_val</b>	Bit pattern representing the current TTL levels; mapping of bits and event inputs see table. Bit = 0: TTL level low. Bit = 1: TTL level high.	LONG

## See also

[P2\\_EVENT\\_ENABLE](#), [P2\\_EVENT\\_CONFIG](#), [P2\\_EVENT2\\_CONFIG](#)

## To be used for the modules

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

## Example

```
#INCLUDE ADwinPRO_ALL.INC

INIT:
REM Configure event input for minimum time of 15 ns,
REM falling edge, 4 edges on module 1
P2_EVENT_READ (1,0,2,4)
REM Enable an external event at the module 1
P2_EVENT_ENABLE (1,1)
```

## P2\_EVENT\_READ

## 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

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

### See also

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

### To be used for the modules

all modules for Pro II bus.

### Example

```
#INCLUDE ADwinPRO_ALL.INC  
INIT:  
    P2_SET_LED(1,1)           'Switch on LED of module 1  
  
EVENT:  
    ...  
  
FINISH:  
    P2_SET_LED(1,0)           'Switch off LED of module 1
```

**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 LONG modules 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.
Analog output	Start D/A conversion on all enabled DACs with the value of the DAC register.

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\_ADCF\_MODE**: Start automatic conversion on several modules.
- **P2\_BURST\_START**: Start burst sequence on several modules.

## See also

[P2\\_SYNC\\_ENABLE](#), [P2\\_SYNC\\_STAT](#)

[P2\\_ADCF\\_MODE](#), [P2\\_BURST\\_START](#), [P2\\_READ\\_ADC](#), [P2\\_READ\\_ADC24](#)

[P2\\_WRITE\\_DAC](#), [P2\\_WRITE\\_DAC4](#), [P2\\_WRITE\\_DAC8\\_PACKED](#), [P2\\_WRITE\\_DAC32](#)

## To be used for the modules

Aln-32/18 Rev. E, Aln-8/18 Rev. E, Aln-F-8/14 Rev. E, Aln-F-8/18 Rev. E, AOut-4/16 Rev. E, AOut-8/16 Rev. E

## P2\_SYNC\_ALL

## Example

```
#INCLUDE ADwinPRO_ALL.INC
DIM i AS LONG
DIM DATA_1[1000], DATA_2[1000], DATA_3[1000] AS LONG
DIM outval[1000] AS LONG

INIT:
    REM Enable synchronization for channels 1+2 on the modules
    REM 1, 2 and 4, disable for all other channels
    P2_SYNC_ENABLE(1,11b)
    P2_SYNC_ENABLE(2,11b)
    P2_SYNC_ENABLE(4,11b)
    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
    DATA_1[i]=P2_READ_ADC(1) 'Read out A/D converter module 1
    DATA_2[i]=P2_READ_ADC(2) 'Read out A/D converter module 2
    DATA_3[i]=P2_READ_ADC(4) 'Read out A/D converter module 4
    REM write value into output register 1 of D/A module 5
    P2_WRITE_DAC(5,1,outval[i])
    IF (i=1000) THEN END             'End process after 1000 repetitions
    INC(i)                           'Increment index
```

**P2\_SYNC\_ENABLE** enables or disables the synchronizing option for selected inputs / outputs on the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.INC

P2_SYNC_ENABLE (module, channel)
```

## Parameters

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

**channel** Bit pattern for selection of inputs / outputs that are enabled or disabled: LONG

0 : disable.  
1: enable.

Bits in <code>channel</code>	31:08	07	06	05	04	03	02	01	00
Channel no. on analog input module	–	8	7	6	5	4	3	2	1
AIIn-F-x/x Rev. E									
Channel no. on analog output module	–	8	7	6	5	4	3	2	1

## Notes

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

The instruction sets all channels 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\\_STAT](#)

## To be used for the modules

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

## P2\_SYNC\_ENABLE

## Example

```
#INCLUDE ADwinPRO_ALL.INC
DIM i AS LONG
DIM DATA_1[1000], DATA_2[1000], DATA_3[1000] AS LONG
DIM outval[1000] AS LONG

INIT:
  REM Enable synchronization for channels 1+2 on the modules
  REM 1, 2 and 4, disable for all other channels
  P2_SYNC_ENABLE(1,11b)
  P2_SYNC_ENABLE(2,11b)
  P2_SYNC_ENABLE(4,11b)
  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
  DATA_1[i]=P2_READ_ADC(1) 'Read out A/D converter module 1
  DATA_2[i]=P2_READ_ADC(2) 'Read out A/D converter module 2
  DATA_3[i]=P2_READ_ADC(4) 'Read out A/D converter module 4
  REM write value into output register of D/A module 5
  P2_WRITE_DAC(5,1,outval[i])
  IF (i=1000) THEN END              'End process after 1000 repetitions
  INC(i)                            'Increment index
```



**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

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

**ret\_val** Setting of the synchronizing option at the LONG inputs / outputs:  
0 : Channel is disabled.  
1 : Channel is enabled.

Bits in channel	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	–	8	7	6	5	4	3	2	1

## Notes

You set the synchronizing option of the channels with **P2\_SYNC\_ENABLE**.

## See also

[P2\\_SYNC\\_ALL](#), [P2\\_SYNC\\_ENABLE](#)

## To be used for the modules

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

## P2\_SYNC\_STAT

## 4.2 Pro II: Input Modules

The include file `ADwinPRO_ALL.INC` includes all functions and procedures that are necessary to get access to the *ADwin-Pro II* A/D-modules. Thus, include the following *ADbasic* command into your program:

```
#INCLUDE ADwinPRO_ALL.INC
```

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

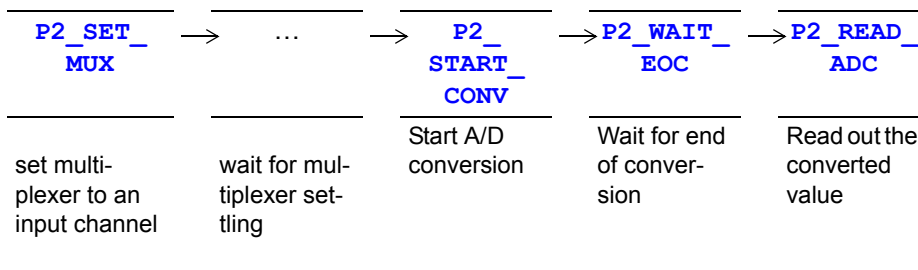
```
#INCLUDE ADwinPRO_ALL.INC
ret_val = P2_ADC(module, input_no)
```

## Parameters

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

## Notes

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



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

## See also

P2\_ADC24, P2\_ADC\_READ\_LIMIT, P2\_ADC\_SET\_LIMIT, P2\_READ\_ADC, P2\_START\_CONV, P2\_WAIT\_EOC

## To be used for the modules

Aln-32/18 Rev. E, Aln-8/18 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 has a resolution of 24 bit.

### Syntax

```
#INCLUDE ADwinPRO_ALL.INC

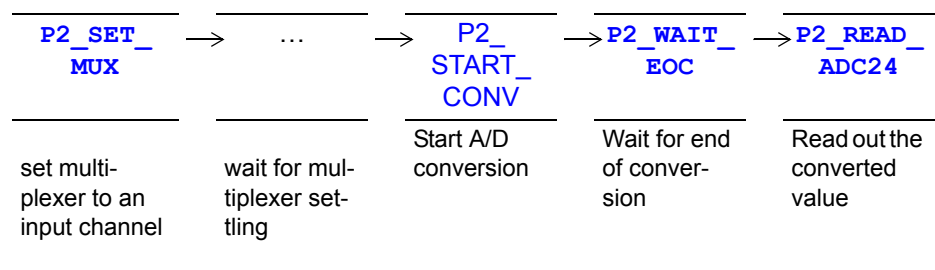
ret_val = P2_ADC24(module, input_no)
```

### Parameters

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

### Notes

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



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

### See also

[P2\\_ADC](#), [P2\\_ADC\\_READ\\_LIMIT](#), [P2\\_ADC\\_SET\\_LIMIT](#), [P2\\_READ\\_ADC24](#), [P2\\_START\\_CONV](#), [P2\\_WAIT\\_EOC](#)

### To be used for the modules

Aln-32/18 Rev. E, Aln-8/18 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 LONG and -underrun flags:

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](#)

## To be used for the modules

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

module	Selected module address (1...15).	LONG
input_no	Number of the analog input (1...8 or 1...32).	LONG
high	Upper limit (0...65535) of the channel. Default: 65535.	LONG
low	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](#)

## To be used for the modules

Aln-32/18 Rev. E, Aln-8/18 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](#)

### To be used for the modules

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

### Example

```
#INCLUDE ADwinPRO_ALL.INC  
DIM value1 AS LONG 'Declaration  
  
EVENT:  
  P2_START_CONV(1,1) 'Start AD conversion  
  P2_WAIT_EOC(1,1) 'Wait for end of conversion  
  value1 = P2_READ_ADC(1,1) '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

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

## Notes

-/-

## See also

[P2\\_ADC24](#), [P2\\_START\\_CONV](#), [P2\\_WAIT\\_EOC](#), [P2\\_ADC\\_READ\\_LIMIT](#), [P2\\_ADC\\_SET\\_LIMIT](#), [P2\\_READ\\_ADC\\_SCONV24](#)

## To be used for the modules

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

## Example

```
#INCLUDE ADwinPRO_ALL.INC
DIM value1 AS LONG 'Declaration

EVENT:
  P2_START_CONV(1,1) 'Start AD conversion
  P2_WAIT_EOC(1,1) 'Wait for end of conversion
  value1 = P2_READ_ADC24(1,1) '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](#)

### To be used for the modules

Aln-32/18 Rev. E, Aln-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_CONV(1,1) 'start A/D conversion

EVENT:
    P2_WAIT_EOC(1,1)
    DATA_1[i] = P2_READ_ADC_SCONV(1,1) '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

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

## See also

[P2\\_ADC](#), [P2\\_ADC24](#), [P2\\_READ\\_ADC](#), [P2\\_READ\\_ADC\\_SCONV](#), [P2\\_START\\_CONV](#), [P2\\_WAIT\\_EOC](#)

## To be used for the modules

Aln-32/18 Rev. E, Aln-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_CONV(1,1) 'start A/D conversion

EVENT:
P2_WAIT_EOC(1,1)
DATA_1[i] = P2_READ_ADC_SCONV24(1,1) 'Read out + start A/D
converter

'24 bit
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

module	Selected module address (1...15).	LONG
choice	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](#)

### To be used for the modules

Aln-32/18 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

<code>module</code>	Selected module address (1...15).	LONG
<code>mode</code>	Operating mode of the sequential control: 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.	LONG
<code>gain</code>	Gain factor (modes 1 ... 3 only): 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.	LONG
<code>channels</code>	Bit pattern to select the channels for conversion. Bit = 0: Don't convert. Bit = 1: Do conversion.	LONG

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

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

## 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`. The end of conversion (for all channels) is automatically synchronized with the beginning of the next process cycle. The end of conversion (for all channels) is automatically synchronized with the beginning of the next 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.

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 setting of 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`

#### To be used for the modules

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

<code>module</code>	Selected module address (1...15).	LONG
<code>count</code>	Even number (2...32) of read measurement values. An odd number is not allowed.	LONG
<code>array[]</code>	Destination array to store the measurement values.	ARRAY LONG FLOAT
<code>array_idx</code>	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\\_READ](#), [P2\\_SEQ\\_READ24](#), [P2\\_SEQ\\_READ\\_PACKED](#), [P2\\_SEQ\\_START](#), [P2\\_SEQ\\_WAIT](#)

### To be used for the modules

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

### Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[16] AS LONG AT DM_LOCAL

INIT:
    P2_SE_DIFF(1,0)           'set inputs to single ended
    REM sequential control: continuous Mode, gain 1
    REM odd-numbered channels, standard settling time
    P2_SEQ_INIT(1,3,0,5555555h,0)
    P2_SEQ_START(1)           'start sequential control
    P2_SEQ_WAIT(1)            'wait until all selected channels
                                'are converted once

EVENT:
    P2_SEQ_READ(1,16,DATA_1,1) 'copy current values from the module
                                'into DATA_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

<code>module</code>	Selected module address (1...15).	LONG
<code>count</code>	Number (1...32) of read measurement values.	LONG
<code>array[]</code>	Destination array to store the measurement values.	ARRAY LONG <del>FLOAT</del>
<code>array_idx</code>	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\\_READ](#), [P2\\_SEQ\\_READ\\_PACKED](#), [P2\\_SEQ\\_START](#), [P2\\_SEQ\\_WAIT](#)

## To be used for the modules

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

## Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[16] AS LONG AT DM_LOCAL

INIT:
  P2_SE_DIFF(1,0)          'set inputs to single ended
  REM sequential control: continuous Mode, gain 1
  REM odd-numbered channels, standard settling time
  P2_SEQ_INIT(1,3,0,55555555h,0)
  P2_SEQ_START(1)          'start sequential control
  P2_SEQ_WAIT(1)           'wait until all selected channels
                           are converted once

EVENT:
  P2_SEQ_READ24(1,16,DATA_1,1) 'copy current values from the
                              'module into DATA_1
```

## P2\_SEQ\_READ24

## P2\_SEQ\_READ\_PACKED

**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

module	Selected module address (1...15).	LONG
count	Number of value pairs to read (1...16).	LONG
array[]	Destination array to store the measurement values.	ARRAY LONG FLOAT
array_idx	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](#)

### To be used for the modules

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

### Example

```
#INCLUDE ADwinPRO_ALL.inc

DIM DATA_1[32], DATA_2[32] AS LONG AT DM_LOCAL

INIT:
  P2_SE_DIFF(5,0)           'set inputs to single ended
  REM modules 1+5: Sequential control continuous mode
  REM gain 1, even-numbered channels
  REM standard settling time
  P2_SEQ_INIT(1,3,0,0AAAAAAAh,0)
  P2_SEQ_INIT(5,3,0,0AAAAAAAh,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, DATA_2
  P2_SEQ_READ_PACKED(1,8,DATA_1,1)
  P2_SEQ_READ_PACKED(5,8,DATA_1,1)
```

**P2\_SEQ\_START** starts the sequence control on all selected modules at once.

## Syntax

```
#INCLUDE ADwinPRO_ALL.inc
P2_SEQ_START(module_pattern)
```

## Parameters

**module\_pattern** 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

## 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](#)

## To be used for the modules

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

## Example

```
#INCLUDE ADwinPRO_ALL.inc
#DEFINE module 4 'Module address

DIM DATA_1[32] AS FLOAT 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\_START

## P2\_SEQ\_WAIT

**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](#)

### To be used for the modules

Aln-32/18 Rev. E, Aln-8/18 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\_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](#)

## To be used for the modules

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

## Example

```
#INCLUDE ADwinPRO_ALL.inc
DIM value1 AS LONG 'Declaration

EVENT:
  P2_SET_MUX(1,0100000010b)'set MUX to input 3, gain to 2
  REM wait for multiplexer settling
  ...
  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\_SET\_MUX

## P2\_START\_CONV

**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](#)

### To be used for the modules

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

### Example

```
#INCLUDE ADwinPRO_ALL.INC  
DIM value AS LONG      'Declaration  
  
EVENT:  
  P2_START_CONV(1,1)      'start AD conversion on channel 1  
  P2_WAIT_EOC(1,1)      'Wait for end of conversion  
  value = P2_READ_ADC(1,1) 'Read value from ADC
```

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

## Syntax

```
#INCLUDE ADwinPRO_ALL.INC
P2_WAIT_EOC (module)
```

## Parameters

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

## Notes

-/-

## See also

[P2\\_ADC](#), [P2\\_ADC24](#), [P2\\_START\\_CONV](#), [P2\\_READ\\_ADC](#)

## To be used for the modules

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

## Example

```
#INCLUDE ADwinPRO_ALL.INC
DIM value AS LONG          'Declaration

EVENT:
  P2_START_CONV(1,1)        'start AD conversion
  P2_WAIT_EOC(1,1)          'Wait for end of conversion
  value = P2_READ_ADC(1,1)  'Read value from ADC
```

## P2\_WAIT\_EOC

## P2\_ADCF

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

### Syntax

```
#INCLUDE ADwinPRO_ALL.INC

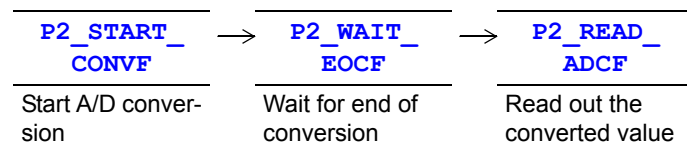
ret_val = P2_ADCF(module, input_no)
```

### Parameters

<code>module</code>	Specified module address (1...15).	LONG
<code>input_no</code>	Number of the analog input (1...4 or 1...8).	LONG
<code>ret_val</code>	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](#)

### To be used for the modules

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

## Syntax

```
#INCLUDE ADwinPRO_ALL.INC

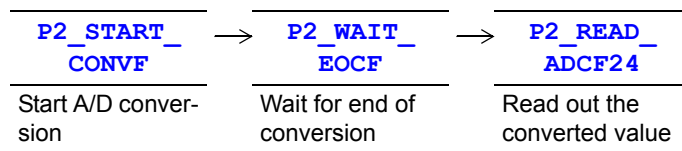
ret_val = P2_ADCF24(module, input_no)
```

## Parameters

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...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, P2\_READ\_ADCF24, P2\_START\_CONVF, P2\_WAIT\_EOCF

## To be used for the modules

AIIn-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\_ADCF24

## P2\_ADCF\_MODE

**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
4	Event Mode
6	Event Mode with Multiplex option

## 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 the instruction **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:

The instruction **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.



## Standard Mode

## Timer Mode

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.

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.

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\\_CONV](#), [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](#)

## To be used for the modules

AIIn-F-8/18 Rev. E



### Timer Mode with Multiplex option

### Event Mode

### Event Mode with Multiplex option



## Example

```
#INCLUDE ADwinPRO_ALL.INC
DIM value[4] AS LONG

INIT:
...
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** 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

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

**ret\_val** 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](#)

## To be used for the modules

Aln-F-8/14 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)
    REM limit-overflow
    ...
  ENDIF
  IF (flags AND 2000h = 2000h)
    REM limit-underrun
    ...
  ENDIF
```

## P2\_ADCF\_READ\_LIMIT

## P2\_ADCF\_SET\_LIMIT

**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

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

### To be used for the modules

Aln-F-8/14 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)
        REM limit-overflow
        ...
    ENDIF
    IF (flags AND 20000h = 20000h)
        REM limit-underrun
        ...
    ENDIF
```

**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](#)

## To be used for the modules

Aln-F-8/14 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\\_CONV](#), [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](#)

### To be used for the modules

Aln-F-8/18 Rev. E

### Example

```
#INCLUDE ADwinPRO_ALL.INC  
DIM value1 AS LONG 'Declaration  
  
EVENT:  
    P2_START_CONV(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

module	Specified module address (1...15).	LONG
array[]	Destination array, where conversion results 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\_ADCF**.

## See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_START\\_CONV](#), [P2\\_READ\\_ADCF](#), [P2\\_READ\\_ADCF8](#), [P2\\_READ\\_ADCF4\\_PACKED](#), [P2\\_READ\\_ADCF8\\_PACKED](#), [P2\\_READ\\_ADCF\\_SCONV](#), [P2\\_WAIT\\_EOCF](#)

## To be used for the modules

Aln-F-8/14 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_CONV(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_CONV(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 <del>FLOAT</del>
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\\_CONV](#), [P2\\_READ\\_ADCF](#), [P2\\_READ\\_ADCF8](#), [P2\\_READ\\_ADCF4\\_PACKED](#), [P2\\_READ\\_ADCF8\\_PACKED](#), [P2\\_READ\\_ADCF\\_SCONV24](#), [P2\\_READ\\_ADCF8\\_24B](#), [P2\\_WAIT\\_EOCF](#)

### To be used for the modules

Aln-F-8/18 Rev. E

### Example

```
#INCLUDE ADWINPRO_ALL.INC
DIM value[4] AS LONG      'Array for conversion results

INIT:
    P2_START_CONV(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_CONV(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

module	Specified module address (1...15).	LONG
array[]	Destination array, where conversion results 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...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\\_CONV](#), [P2\\_WAIT\\_EOCF](#), [P2\\_READ\\_ADCF4](#), [P2\\_READ\\_ADCF4\\_PACKED](#), [P2\\_READ\\_ADCF8\\_PACKED](#), [P2\\_READ\\_ADCF\\_SCONV](#)

## To be used for the modules

Aln-F-8/14 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_CONV(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_CONV(1,0FFh)   'Start new AD conversion
```

## P2\_READ\_ADCF8



**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

module	Specified module address (1...15).	LONG
array[]	Destination array, where conversion results 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 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
index	F-ADC 2	F-ADC 1
index+1	F-ADC 4	F-ADC 3

## See also

P2\_ADCF, P2\_ADCF24, P2\_START\_CONV, P2\_WAIT\_EOCF, P2\_READ\_ADCF4, P2\_READ\_ADCF8, P2\_READ\_ADCF8\_PACKED, P2\_READ\_ADCF\_SCONV

## To be used for the modules

Aln-F-8/14 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_CONV(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_CONV(1,0Fh) 'Start new AD conversion
```

## P2\_READ\_ADCF4\_PACKED



**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\\_READ\\_ADCF](#), [P2\\_READ\\_ADCF4](#), [P2\\_READ\\_ADCF8](#), [P2\\_READ\\_ADCF4\\_PACKED](#), [P2\\_READ\\_ADCF8\\_PACKED](#), [P2\\_READ\\_ADCF\\_SCONV](#), [P2\\_READ\\_ADCF\\_SCONV32](#), [P2\\_START\\_CONV](#), [P2\\_WAIT\\_EOCF](#)

## To be used for the modules

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

## Example

```
#INCLUDE ADwinPRO_ALL.INC
DIM value1 AS LONG 'Declaration

EVENT:
  P2_START_CONV(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

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

### 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](#)

### To be used for the modules

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

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

## 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\_CONV, P2\_WAIT\_EOCF

## To be used for the modules

AIIn-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_CONV(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 LONG 1...4.

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](#)

### To be used for the modules

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\_START\_CONV** starts the conversion on one or more F-ADCs of the specified module.

## Syntax

```
#INCLUDE ADwinPRO_ALL.INC

P2_START_CONV(module, adc_no)
```

## Parameters

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

**adc\_no** 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.

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](#)

## To be used for the modules

AIIn-F-8/18 Rev. E

## Example

```
#INCLUDE ADwinPRO_ALL.INC
DIM value AS LONG 'Declaration

EVENT:
  P2_START_CONV(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\_START\_CONV

## P2\_WAIT\_EOCF

**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).	<span style="border: 1px solid black; padding: 0 2px;">LONG</span>
<b>adc_no</b>	Bit pattern that determines the ADCs, whose end of conversion shall be awaited (see table).	<span style="border: 1px solid black; padding: 0 2px;">LONG</span>

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.

### See also

[P2\\_ADCF](#), [P2\\_ADCF24](#), [P2\\_START\\_CONV](#), [P2\\_READ\\_ADCF](#), [P2\\_READ\\_ADCF4](#), [P2\\_READ\\_ADCF8](#), [P2\\_READ\\_ADCF4\\_PACKED](#), [P2\\_READ\\_ADCF8\\_PACKED](#)

### To be used for the modules

Aln-F-8/18 Rev. E

### Example

```
#INCLUDE ADwinPRO_ALL.INC
DIM value AS LONG 'Declaration

EVENT:
  P2_START_CONV(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\_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

<code>module</code>	Specified module address (1...15).	LONG
<code>count</code>	Amount of measurement values to be transferred. The amount must be divisible by 8.	LONG
<code>array[]</code>	Destination array, where the measurement values are transferred. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
<code>array_idx</code>	Destination start index: Array element from which measurement values are stored.	LONG
<code>flowrate</code>	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 the instruction **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.

## P2\_BURST\_CREAD\_UNPACKED1



## See also

[P2\\_BURST\\_INIT](#), [P2\\_BURST\\_CREAD\\_UNPACKED2](#), [P2\\_BURST\\_CREAD\\_UNPACKED4](#), [P2\\_BURST\\_CREAD\\_UNPACKED8](#), [P2\\_BURST\\_START](#), [P2\\_BURST\\_STATUS](#)

## To be used for the modules

AIIn-F-8/14 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** 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

<code>module</code>	Specified module address (1...15).	LONG
<code>count</code>	Amount of measurement values per channel to be transferred. The amount must be divisible by 4.	LONG
<code>arrayx[]</code>	Destination arrays for the measurement values of channels 1 and 2. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
<code>array_idx</code>	Destination start index: Array element from which measurement values are stored.	LONG
<code>flowrate</code>	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 the instruction **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.

## P2\_BURST\_CREAD\_UNPACKED2



## See also

[P2\\_BURST\\_INIT](#), [P2\\_BURST\\_CREAD\\_UNPACKED1](#), [P2\\_BURST\\_CREAD\\_UNPACKED4](#), [P2\\_BURST\\_CREAD\\_UNPACKED8](#), [P2\\_BURST\\_START](#), [P2\\_BURST\\_STATUS](#)

## To be used for the modules

AIIn-F-8/14 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** 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

<code>module</code>	Specified module address (1...15).	LONG
<code>count</code>	Amount of measurement values per channel to be transferred. The amount must be divisible by 2.	LONG
<code>arrayx[]</code>	Destination arrays for the measurement values of channels 1...4. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
<code>array_idx</code>	Destination start index: Array element from which measurement values are stored.	LONG
<code>flowrate</code>	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 the instruction **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.

## P2\_BURST\_CREAD\_UNPACKED4



## See also

[P2\\_BURST\\_INIT](#), [P2\\_BURST\\_CREAD\\_UNPACKED1](#), [P2\\_BURST\\_CREAD\\_UNPACKED2](#), [P2\\_BURST\\_CREAD\\_UNPACKED8](#), [P2\\_BURST\\_START](#), [P2\\_BURST\\_STATUS](#)

## To be used for the modules

AI-F-8/14 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** 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

<code>module</code>	Specified module address (1...15).	LONG
<code>count</code>	Amount of measurement values per channel to be transferred. The amount of values must be divisible by 4.	LONG
<code>arrayx[]</code>	Destination arrays for the measurement values of channels 1...8. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
<code>array_idx</code>	Destination start index: Array element from which measurement values are stored.	LONG
<code>flowrate</code>	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 the instruction **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.

## P2\_BURST\_CREAD\_UNPACKED8



## See also

[P2\\_BURST\\_INIT](#), [P2\\_BURST\\_CREAD\\_UNPACKED1](#), [P2\\_BURST\\_CREAD\\_UNPACKED2](#), [P2\\_BURST\\_CREAD\\_UNPACKED4](#), [P2\\_BURST\\_START](#), [P2\\_BURST\\_STATUS](#)

## To be used for the modules

AIIn-F-8/14 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** 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. LONG

In combination with the memory size the maximum amount of measurement values per channel is set:

chan- nels	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:

chan- nels	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.

## P2\_BURST\_INIT

## Operating mode

## Clock speed

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

Bit no.	03...31	02	01	00
Func- tion	–	Type of clock speed: Bit = 0: Timer controlled ( <i>pulses</i> ). 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 time-line 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.

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.

### See also

**P2\_BURST\_CREAD\_UNPACKED1**, **P2\_BURST\_CREAD\_UNPACKED2**, **P2\_BURST\_CREAD\_UNPACKED4**, **P2\_BURST\_CREAD\_UNPACKED8**, **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**

### To be used for the modules

AIIn-F-8/14 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

**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

<code>module</code>	Specified module address (1...15).	LONG
<code>ret_val</code>	Address (0...268435452 = 2 <sup>28</sup> - 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\_READ\_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](#)

### To be used for the modules

AIIn-F-8/14 Rev. E



## 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

**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

module	Specified module address (1...15).	LONG
count	Amount of 32-bit values to be transferred The amount must be divisible by 4.	LONG
startadr	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
array[]	Destination array, where the measurement values are transferred. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
array_idx	Destination start index: Array element from which measurement values are stored.	LONG
flowrate	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
startadr	C1 / D2	C1 / D1
startadr+1	C1 / D4	C1 / D3
startadr+2	C1 / D6	C1 / D5
...	...	...

No. of channels: 1

address	Bits 31:16	Bits 15:00
startadr	C2 / D1	C1 / D1
startadr+1	C2 / D2	C1 / D2
startadr+2	C2 / D3	C1 / D3
...	...	...

No. of channels: 2

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 troughput—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](#)

To be used for the modules

AI{n}-F-8/14 Rev. E

Example

see [P2\\_BURST\\_READ\\_INDEX](#)



## P2\_BURST\_READ\_UNPACKED1

**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,  
array1[], array_idx, flowrate)
```

### Parameters

<code>module</code>	Specified module address (1...15).	LONG
<code>count</code>	Amount of measurement values to be transferred. The amount must be divisible by 8.	LONG
<code>startadr</code>	Start address ( $0 \dots 2^{28} - 4$ ) in the module memory: Address from which the measurement values are read. The address must be divisible by 4.	LONG
<code>array1[]</code>	Destination array, where the measurement values are transferred. No float type and no FIFO array allowed.	ARRAY LONG FLOAT
<code>array_idx</code>	Destination start index: Array element from which measurement values are stored.	LONG
<code>flowrate</code>	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](#)

### To be used for the modules

Aln-F-8/14 Rev. E



## Example

See also examples for [Continuous signal conversion](#) on page 335.

```
#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

**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

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

### To be used for the modules

Aln-F-8/14 Rev. E



## Example

See also examples for [Continuous signal conversion](#) on page 335.

```
#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

**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

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

### To be used for the modules

Aln-F-8/14 Rev. E





## Example

See also examples for [Continuous signal conversion](#) on page 335.

```
#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

**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

<code>module</code>	Specified module address (1...15).	LONG
<code>count</code>	Amount of measurement values to be transferred.	LONG
<code>startadr</code>	Start address (0...268435452 = $2^{28} - 4$ ) in the module memory: Address from which the measurement values are read. The address must be divisible by 4.	LONG
<code>arrayx[]</code>	Destination arrays for the measurement values of channels 1...8.	ARRAY LONG FLOAT
<code>array_idx</code>	Destination start index: Array element from which measurement values are stored.	LONG
<code>flowrate</code>	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](#)

### To be used for the modules

Aln-F-8/14 Rev. E



## Example

See also examples for [Continuous signal conversion](#) on page 335.

```
#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

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

LONG

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](#)

### To be used for the modules

Aln-F-8/14 Rev. E



## Example

```
#INCLUDE ADwinPRO_ALL.INC
#DEFINE module 4

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 module 4 only
  P2_BURST_START(pattern)
  PROCESSDELAY=10000000
  state=0

EVENT:
  REM Anzahl der restlichen Messwerte holen
  rest = P2_BURST_STATUS(module)
  REM Alle Messwerte liegen vor: Status ändern
  IF (rest=0) THEN state=1
  IF (state=1) THEN
    REM Alle Messwerte liegen vor: 1000 Messwerte (schnell)
    REM abholen und in DATA_1 ablegen
    P2_BURST_READ_UNPACKED1(module,1000,0,DATA_1,1,3)
    REM Nächste Burst-Messreihe starten
    state=0
    P2_BURST_RESET(pattern)
    P2_BURST_START(pattern)
  ENDIF
```

## P2\_BURST\_START



**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\_pattern** 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

### 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](#)

### To be used for the modules

Aln-F-8/14 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\_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

module	Specified module address (1...15).	LONG
ret_val	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](#)

## To be used for the modules

Aln-F-8/14 Rev. E

## P2\_BURST\_STATUS

### 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** 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\_pattern** Bit pattern for accessing the module addresses: LONG  
 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](#)

## To be used for the modules

Aln-F-8/14 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 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)
      P2_BURST_STOP(pattern)
    ENDIF
  NEXT
```

## P2\_BURST\_STOP

## P2\_SET\_AVERAGE\_FILTER

**P2\_SET\_AVERAGE\_FILTER** determines if the module calculates a moving 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 over the last converted measurement values.

### See also

[P2\\_BURST\\_INIT](#), [P2\\_BURST\\_READ\\_UNPACKED1](#), [P2\\_BURST\\_CREAD\\_UNPACKED1](#), [P2\\_READ\\_ADCF](#)

### To be used for the modules

Aln-F-8/14 Rev. E

### Example

```
#INCLUDE ADwinPRO_ALL.INC  
  
INIT:  
  REM Initiate filter to average over 2 samples  
  P2_SET_AVERAGE_FILTER(1,1)
```

### 4.3 Pro II: Output Modules

The include file `ADwinPRO_ALL.INC` includes all functions and procedures that are necessary to get access to the *ADwin-Pro II* D/A-modules. Thus, include the following *ADbasic* command into your program:

```
#INCLUDE ADwinPRO_ALL.INC
```

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

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



## P2\_DAC

**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)
```

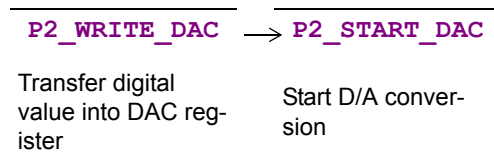
### Parameters

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.

The instruction **P2\_DAC** is characterized by a sequence of several commands:



### 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](#)

### To be used for the modules

AOut-4/16 Rev. E, AOut-8/16 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\_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

module	Specified module address (1...15).	LONG
array[]	Array with values (0...65535) to be output.	ARRAY LONG <del>FLOAT</del>
index	Index of the first array element to be output.	LONG

## Notes

The instruction **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

## To be used for the modules

AOut-4/16 Rev. E, AOut-8/16 Rev. E

## Example

```
REM Digital proportionl controller for 4 channels
#include ADWINPRO_ALL.INC
DIM sp, dev AS LONG
DIM 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

**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

<code>module</code>	Specified module address (1...15).	LONG
<code>array[]</code>	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
<code>index</code>	Index ( <sup>31</sup> ) of the first array element to be output.	LONG

### Notes

The instruction **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	<code>array[n+1]</code>		<code>array[n]</code>	
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](#)

### To be used for the modules

AOut-4/16 Rev. E, AOut-8/16 Rev. E

## Example

```
REM Digital proportionl controller for 4 channels
#include ADwinPRO_ALL.INC
DIM sp, dev1, dev2 AS LONG
DIM 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 FFh)
    dev2 = sp - (SHIFT_RIGHT(array[i],16) AND FFh)
    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

**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

<code>module</code>	Specified module address (1...15).	LONG
<code>array[]</code>	Array with values (0...65535) to be output.	ARRAY LONG FLOAT
<code>index</code>	Index ( <sup>31</sup> ) of the first array element to be written.	LONG

### Notes

The instruction **P2\_DAC8** is characterized by a sequence of several commands:

<b>P2_WRITE_DAC8</b>	→	<b>P2_START_DAC</b>
8 Digitalwerte in das DAC-Register übertragen.		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

### To be used for the modules

AOut-8/16 Rev. E

### Example

```
REM Digital proportionl controller for 4 channels
#include ADWINPRO_ALL.INC
DIM sp, dev AS LONG
DIM 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\_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

module	Specified module address (1...15).	LONG
array[]	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
index	Index ( <sup>31</sup> ) of the first array element to be written.	LONG

## Notes

The instruction **P2\_DAC8\_PACKED** is characterized by a sequence of two commands:

<b>P2_WRITE_DAC8</b>	→	<b>P2_START_DAC</b>
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](#)

## To be used for the modules

AOut-8/16 Rev. E

## P2\_DAC8\_PACKED

### Example

```
REM Digital proportionl controller for 8 channels
#include ADWINPRO_ALL.INC
DIM sp, dev1, dev2 AS LONG
DIM 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 FFh)
    dev2 = sp - (SHIFT_RIGHT(array[i],16) AND FFh)
    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** 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 the instruction **P2\_SYNC\_ALL**.

## See also

[P2\\_DAC](#), [P2\\_DAC4](#), [P2\\_DAC4\\_PACKED](#), [P2\\_DAC8](#), [P2\\_DAC8\\_PACKED](#), [P2\\_SYNC\\_ALL](#), [P2\\_WRITE\\_DAC](#), [P2\\_WRITE\\_DAC4](#), [P2\\_WRITE\\_DAC4\\_PACKED](#), [P2\\_WRITE\\_DAC8](#), [P2\\_WRITE\\_DAC8\\_PACKED](#), [P2\\_WRITE\\_DAC32](#)

## To be used for the modules

AOut-4/16 Rev. E, AOut-8/16 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\_START\_DAC

## P2\_WRITE\_DAC

**P2\_WRITE\_DAC** a digital value into the output register of a DAC on the specified module.

The instruction **P2\_START\_DAC** starts the conversion of the digital value into an output voltage.

### 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

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

### To be used for the modules

AOut-4/16 Rev. E, AOut-8/16 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\_DAC4** writes 4 digital values from an array into the output registers of the DAC 1...4 of the specified module.

The instruction **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

module	Specified module address (1...15).	LONG
array[]	Array with values (0...65535) to be output.	ARRAY LONG <del>FLOAT</del>
index	Index des ersten auszugebenden Feldelements.	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

## To be used for the modules

AOut-4/16 Rev. E, AOut-8/16 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

## P2\_WRITE\_DAC4\_PACKED

**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.

The instruction **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

module	Specified module address (1...15).	LONG
array[]	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
index	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

### To be used for the modules

AOut-4/16 Rev. E, AOut-8/16 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\_DAC8** writes 8 digital values from an array into the output registers of the DAC 1...8 of the specified module.

The instruction **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

module	Specified module address (1...15).	LONG
array[]	Array with values (0...65535) to be output.	ARRAY LONG <del>FLOAT</del>
index	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

## To be used for the modules

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

## P2\_WRITE\_DAC8\_PACKED

**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.

The instruction **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

module	Specified module address (1...15).	LONG
array[]	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
index	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

### To be used for the modules

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\_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 the instruction **START\_DAC**.

### Syntax

```
#INCLUDE ADWINPRO_ALL.INC

P2_WRITE_DAC32 (module, dac_no, value32)
```

### Parameter

<code>module</code>	Specified module address (1...15).	LONG
<code>dac_no</code>	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
<code>value32</code>	Auszugebender Wert (0h...0FFFFFFFFh).	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](#)

### To be used for the modules

AOut-4/16 Rev. E, AOut-8/16 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
```

## P2\_WRITE\_DAC32

## 5 Program Examples

The following examples are available:

- [Online Evaluation of Measurement Data, Seite 326](#)
- [Digital Proportional Controller, Seite 326](#)
- [Data Exchange with DATA arrays, Seite 327](#)
- [Digital PID Controller, Seite 327](#)
- [Data exchange with fieldbus, Seite 330](#)
- [Examples for RS232 and RS485:](#)
  - [RS232: Send and receive, Seite 331](#)
  - [RS232: Send string instruction, Seite 331](#)
  - [RS232: Receive string instruction, Seite 332](#)
  - [RS485: Receive and send, Seite 334](#)
- [Continuous signal conversion: Convert 1 channel, Seite 335](#)

Most examples are stored as program files in the directory <C:\ADwin\ADbasic\samples\_ADwin\_PRO>.

### 5.1 Online Evaluation of Measurement Data

The program <PRO\_DMO1.BAS> searches for the maximum and minimum value out of 1000 measurements of ADC1 and writes the result to the variables `PAR_1` und `PAR_2`.

You need a 16-bit A/D module with module address 1 (module group AD) and an input signal at channel 1 of the module.

```
#INCLUDE ADwinPRO_ALL.inc 'Include-file for Pro A/D modules
#DEFINE limit 65535      'max. 16 bit ADC-value
DIM i1, iw, max, min AS INTEGER

INIT:
  i1 = 1                  'reset sample counter
  max = 0                 'initial maximum value
  min = limit             'initial minimum value
  PAR_10 = 0             'init End-Flag
  GLOBALDELAY = 40000    'cycle-time of 1ms (T9)

EVENT:
  iw = ADC16(1,1)        'get sample
  IF (iw > max) THEN max = iw 'new maximum sample?
  IF (iw < min) THEN min = iw 'new minimum sample?
  INC i1                 'increment index
  IF (i1 > 1000) THEN
    i1 = 1               'reset index
    PAR_1 = min          'write minimum value
    PAR_2 = max          'write maximum value
    max = 0              'reset minimum value
    min = 65535          'reset maximum value
    ' ACTIVATE_PC        'only for use with TestPoint
    PAR_10 = 1           'set End-Flag
  ENDIF
```

### 5.2 Digital Proportional Controller

The program <PRO\_DMO2.BAS> is a digital proportional controller. The set-point is specified by `PAR_1`, the gain by `PAR_2`.

You need:

- 16-bit A/D module with module address 1 (module group AD).
- D/A module with module address 1 (module group DA).
- An external controlled system that receives the signal from the D/A module and returns a signal to the A/D module.

```
#INCLUDE ADwinPRO_ALL.inc 'Include file for Pro A/D modules
#include ADwinPRO_ALL.inc 'Include file for Pro D/A modules

#define offset 32768      '0V for 16 bit ADC/DAC-systems

REM cd: control deviation; av: actuating value
DIM cd, av AS INTEGER

INIT:
    PAR_1 = offset          'initial setpoint
    PAR_2 = 10              'initial gain
    GLOBALDELAY = 40000     'cycle-time of 1ms (T9)

EVENT:
    cd = PAR_1 - ADC16(1, 1) 'compute control deviation (cd)
    av = cd * PAR_2 + offset 'compute actuating value (av)
    DAC(1, 1, av)           'output actuating value on DAC-#1
```

## 5.3 Data Exchange with DATA arrays

The program <PRO\_DMO3.BAS> measures the analog input 1 of the A/D module with address 1 and sets an end flag after 1000 measurements to indicate that the computer can now get the measurement data. The data are transferred by using the array DATA\_1.

You need a 16-bit A/D module with module address 1 (module group AD).

```
#INCLUDE ADwinPRO_ALL.inc 'Include-file for Pro A/D modules

DIM DATA_1[1000] AS INTEGER
DIM index AS INTEGER

INIT:
    PAR_10 = 0
    index = 0          'reset array pointer
    GLOBALDELAY = 40000 'cycle-time of 1ms (T9)

EVENT:
    index = index + 1    'increment array pointer
    IF (index > 1000) THEN '1000 samples done?
        ' ACTIVATE_PC    'set ACTIVATE_PC flag (only necessary
                        'for TestPoint)
        PAR_10 = 1      'set End-Flag
        END             'terminate process
    ENDIF
    DATA_1[index] = ADC16(1, 1) 'acquire sample and save in array
```

## 5.4 Digital PID Controller

The program <PRO\_DMO6.BAS> is a digital PID controller.

Before starting the PID controller the global variables must be set to the controller's values.



You need:

- 16-bit A/D module with module address 1 (module group AD).
- D/A module with module address 1 (module group DA).
- An external controlled system that receives the signal from the D/A module and returns a signal to the A/D module.

#### Calculation on the PC:

The control coefficients are calculated on the computer and transferred as global variables to the processor of the *ADwin-Pro* system. Vice versa the information is returned from the program to the PC.

##### Controller parameter settings

FPAR_2	gain of the controller
FPAR_3	integration time of the controller
FPAR_4	differentiation time of the controller
PAR_1	Setpoint in digits
PAR_6	controller sampling rate in units of 25ns

##### Information from the program

PAR_5	array index (of DATA_1) of control deviation
PAR_9	mean value of control deviation
PAR_10	Flag: All samples are done
DATA_1 [ ]	Array holding all control deviations

#### ADbasic Program:

Both the addresses of the analog input and analog output module are set to the address 1 in the example.

Please note that you will get a time saving effect, when calculation and output of the actuating value is executed during the necessary waiting period during reading the control deviation (after **SET\_MUX** and **START\_CONV**).

The consequence is that the output actuating value is calculated from the control deviation of the previous process call.

```
#INCLUDE ADwinPRO_ALL.inc 'Include file for Pro A/D modules
#INCLUDE ADwinPRO_ALL.inc 'Include file for Pro D/A modules

#DEFINE offset 32768      '0V output

DIM DATA_1[4000] AS LONG
DIM av, cd, cdo, sum AS LONG
DIM diff AS FLOAT

INIT:
    sum = 0                'initial value of integral part
    cd = ADC(1)            'initial value of control deviation
                           '(cd) & MUX to Ch-#1
    PAR_5 = 1              'set array index
    IF (FPAR_3 < 1E3) THEN FPAR_3 = 1E3 'check min. of integration
                           'time
    IF (PAR_6 < 4E4) THEN PAR_6 = 4E4 'allow cycle-times >= 1ms
    GLOBALDELAY = PAR_6    'set cycle-time

EVENT:
    REM compute actuating value
    av = FPAR_2 * (cd + sum / FPAR_3 + diff * FPAR_4)
    START_CONV(1)          'start conversion ADC-#1
    REM while conversion is running ...
    DAC(1, av + offset)    'output actuating value at DAC-#1
    cdo = cd               'keep control deviation in mind
    WAIT_EOC(1)            'wait until end-of-conversion of ADC
    cd = PAR_1 - READADC(1) 'compute control deviation
    FPAR_9 = FPAR_9 * 0.99 + cd * 0.01 'mean value of control
                                   'deviation
    sum = sum + cd         'calculate integral
    IF (sum > 2E6) THEN sum = 2E6 'positive limit of integral
    IF (sum < -2E6) THEN sum = -2E6 'negative limit of integral
    diff = (cd - cdo)      'calculate deviation difference
    DATA_1[PAR_5] = cd    'write control deviation in a buffer
    INC PAR_5              'increment buffer index
    IF (PAR_5 >= 4000) THEN '4000 samples done?
    ' ACTIVATE_PC          'only for use with TestPoint
    PAR_10 = 1             'set End-flag
    PAR_5 = 1              'reset array index
    ENDIF

FINISH:
    DAC(1,offset)          'analog output #1 to 0V
    ,
```

## 5.5 Data exchange with fieldbus

The following program is an example for exchanging data with the fieldbus. It takes into consideration that initializing the fieldbus interface has already been made.

Cyclically (timer-controlled), access to the DP-RAM is requested, the access right is checked, data are exchanged and access is returned again to the bus side. Before the exchange, the data is checked if it has been modified and only then will it be read out and transferred again.

You need:

- Fieldbus module with module address 1 (module group EXT).
- A fieldbus application or device, which can receive and send data.

```
#INCLUDE ADwinPRO_ALL.inc
DIM DATA_1[1000] AS LONG 'Field for input data
DIM DATA_2[1000] AS LONG 'Field for output data
DIM n AS LONG

INIT:
  n = 1
  PAR_14 = 0
  FOR n = 1 TO 100      'Initialization of send data
    DATA_2[n] = n
  NEXT n

EVENT:
  IF (PAR_14 = 0) THEN
    INC PAR_8
    PAR_8 = PAR_8 AND 0ffh
    REQUEST_ACCESS(0,6) 'request access to input and
                        'output areas

    PAR_14 = 1
  ENDIF
  IF (PAR_14 = 1) THEN
    PAR_1 = CHECK_ACCESS(0) 'Check access right
    IF (PAR_1 = 6) THEN 'If it has access right....
      PAR_14 = 2
    ELSE
      REQUEST_ACCESS(0,6) 'else, request once more
    ENDIF
  ENDIF
  IF (PAR_14 = 2) THEN
    PAR_9 = CHANGED_DATA(0,32) 'Check if there are new data
    IF (PAR_9 <> 0) THEN 'If there are new data,then...
      INC PAR_7
      DATA_2[1] = PAR_8
      SET_WRITE_BUFFER(0,DATA_2,0,60) 'Write data (60 bytes)
      GET_READ_BUFFER(0,DATA_1,0200h,40) 'Read data (40 bytes)
    ENDIF
    REQUEST_RELEASE_ACCESS(0,6) 'Release access rights
    PAR_14 = 3
  ENDIF
  IF (PAR_14 = 3) THEN
    PAR_1 = CHECK_ACCESS(0) 'Has the bus access rights again?
    IF (PAR_1 = 0) THEN
      INC PAR_11      'Count access cycles
      PAR_14 = 0
    ENDIF
  ENDIF
```

## 5.6 Examples for RS232 and RS485

The following examples are complete programs for sending and receiving data and strings with RS232 or RS485.

You need a RS232 module with module address 1 (module group EXT).

The following program illustrates the initialization of the serial RS232 interface in the **INIT**: section and cyclic data read and write in the **EVENT**: section. The process is timer-controlled.

```
REM The program initializes the serial interface
REM in the section Init:
REM In the section Event: data is exchanged between the interfaces
REM 1 & 2 of the RS module.
REM The interfaces are tested with this program.
REM For this you have to connect the interfaces with
REM each other before starting the program.
```

```
#INCLUDE ADwinPRO_ALL.inc
DIM DATA_1[1000] AS LONG 'Transmitted data
DIM DATA_2[1000] AS LONG 'Received data
DIM i AS LONG             'Control variable

INIT:
  FOR i = 1 TO 1000        'Initialization of the
                           'transmitted data
    DATA_1[i] = i AND 0FFh
  NEXT i
  RS_INIT(1,1,9600,0,8,1,0) 'Initializing interface 1:
                           '9600 Baud;
                           'No parity bit;
                           '8 data bits;
                           '2 stop bits;
                           'No handshake

  RS_INIT(1,2,9600,0,8,1,0) 'Initializing interface 2
                           'same as interface 1

  PAR_1 = 1
  PAR_4 = 1

EVENT:
  REM Read and write a data set
  IF (PAR_1 <= 1000) THEN 'Send data
    PAR_2 = WRITE_FIFO(1,1,DATA_1[PAR_1])
    IF (PAR_2 = 0) THEN INC PAR_1
  ENDIF

  PAR_3 = READ_FIFO(1,2) 'Read data
  IF (PAR_3 <> -1) THEN
    DATA_2[PAR_4] = PAR_3
    INC PAR_4
  ENDIF
  IF (PAR_4 > 1000) THEN END 'All data are transmitted
```

You need a RS232 module with module address 1 (module group EXT).

Many devices with an RS232 interface can be controlled using string instructions. The following 2 programs show how to send a string in one process and how to receive the string with another process. Both programs are available on the ADwin CDROM.

The programs can be used on the same module but with different interfaces. Please pay attention to the remarks in the programs.

The program RS232\_send\_string.BAS first initializes interface 1. In the **EVENT** section the interface 1 sends a string char by char. In the **FINISH** sec-

**RS232:**  
**Send and receive**

**RS232:**  
**Send string instruction**

tion the character "#" is used as an end marker. It may be replaced by any other character.

```
' Process for RS232-communication: sending a string
' ++++++
' The program may run together with RS232_receive_string.BAS
' on the same module. If so, please follow these instructions:
' - connect the interfaces with each other
' - compile and start RS232_receive_string.BAS
' - compile and start RS232_send_string.BAS

#INCLUDE adwinpro.inc
#INCLUDE ADwinPRO_ALL.inc

REM import string library
#IF PROZESSOR = T10 THEN
IMPORT string.lia
#ELSE
IMPORT string.li9
#ENDIF

#DEFINE rs_adr 1          'module address
#DEFINE rs_no 1           'interface number
#DEFINE s_endchar "#"     'end marker "#"
#DEFINE s_send DATA_1
#DEFINE str_len 50        'length of send string

DIM s_send[str_len] AS STRING 'send string
DIM s_temp[1] AS STRING 'single char
DIM sp AS LONG             'send pointer

INIT:
GLOBALDELAY = 10000000 '0.25 s
'A reset is allowed only once on a module!
'RS_RESET(rs_adr)      'reset RS module
RS_INIT(rs_adr,rs_no,9600,0,8,0,0) 'init RS interface
sp=1                    'initialize pointer
s_send = "This is a TESTSTRING" 'send string

EVENT:
STRMID(s_send, sp, 1, s_temp) 'read next char of string
PAR_11 = ASC(s_temp) 'get ascii code of char
IF (PAR_11 = 0) THEN END 'quit when all chars are sent
PAR_12 = WRITE_FIFO(rs_adr, rs_no, PAR_11) 'send code
REM increase pointer, else send again
IF (PAR_12 = 0) THEN INC sp
REM quit when max. string length is reached
IF (sp > str_len) THEN END

FINISH:
DO
    'send end marker "#"
    PAR_11 = ASC(s_endchar) 'get ascii code
    PAR_12 = WRITE_FIFO(rs_adr, rs_no, PAR_11) 'send code
UNTIL (PAR_12 = 0)
```

## RS232: Receive string instruction

You need a RS232 module with module address 1 (module group EXT).

The program RS232\_receive\_string.BAS first initializes interface 2. In the **EVENT** section the interface 2 receives a string until the end marker char is received (or the receiving string is full)



```
' Process for RS232-communication: Receiving a string.
' ++++++
' The program may run together with RS232_send_string.BAS
' on the same module. If so, please follow these instructions:
' - connect the interfaces with each other
' - compile and start RS232_receive_string.BAS
' - compile and start RS232_send_string.BAS

#include adwinpro.inc
#include ADwinPRO_ALL.inc

REM import string library
#if PROZESSOR = T10 THEN
IMPORT string.lia
#else
IMPORT string.li9
#endif

#define rs_adr 1          'module address
#define rs_no 2           'interface number
#define s_receive DATA_2
#define str_len 50        'max. length of received string

DIM s_receive[str_len] AS STRING 'received string
DIM s_temp[1] AS STRING 'single char
DIM s_endchar[1] AS STRING 'end marker
DIM endflag AS LONG        '
DIM rp AS LONG             'receive pointer

INIT:
GLOBALDELAY = 10000000 '0.25 s
RS_RESET(rs_adr)        'reset RS module
RS_INIT(rs_adr,rs_no,9600,0,8,0,0) 'init RS interface
rp = 0                  'initialize receive pointer
s_receive = ""          'initialize receive string
s_endchar = "#"         'end marker

EVENT:
PAR_21 = READ_FIFO(rs_adr, rs_no) 'receive status / char
IF (PAR_21 <> -1) THEN
  CHR(PAR_21,s_temp) 'get char from ascii value
  INC rp            'increase receive pointer
  REM end marker received or string full?
  endflag = STRCOMP(s_temp, s_endchar)
  IF ((endflag=0) OR (rp>str_len)) THEN END
  s_receive = s_receive + s_temp 'save char to string
ENDIF
```

**RS485:**  
**Receive and send**

You need a RS485 module with module address 1 (module group EXT).

In this example the RS485 interface 2 is a passive participant, which reads data coming from the bus. If a specified byte (55) is received, the interface becomes active and starts sending the value 44.

```
REM This program implies a RS485 interface with the address 1.
#include ADwinPRO_ALL.inc

#define rs_adr 1

DIM ret_val AS LONG
DIM val AS LONG

INIT:
    RS_RESET(rs_adr)
    RS_INIT(rs_adr,2,38400,0,8,0,3) 'Initialize channel 2
    RS485_SEND(rs_adr,2,0) 'channel 2 = receiving

EVENT:
    val = READ_FIFO(rs_adr,2) 'Read data

    IF (val = 55) THEN
        RS485_SEND(rs_adr,2,1) 'channel 2 = sending
        ret_val = WRITE_FIFO(rs_adr,2,44) 'Write data
    ENDIF
```

## 5.7 Continuous signal conversion

The modules Pro II AIn-F-4/14 Rev. E and Pro II AIn-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.

### Convert 1 channel

You need

- one module Pro II AIn-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

```

## Annex

### A.1 Alphabetic Instruction List

#### A

ADC · 18  
P2\_ADC · 239  
ADC16 · 20  
P2\_ADC24 · 240  
ADCF · 22  
P2\_ADCF · 260  
P2\_ADCF24 · 261  
P2\_ADCF\_MODE · 262  
P2\_ADCF\_READ\_LIMIT · 265  
P2\_ADCF\_SET\_LIMIT · 266  
P2\_ADC\_READ\_LIMIT · 241  
P2\_ADC\_SET\_LIMIT · 243

#### B

BURST\_ABORT · 23  
BURST\_CREAD · 25  
P2\_BURST\_CREAD\_UNPACKED1 · 281  
P2\_BURST\_CREAD\_UNPACKED2 · 283  
P2\_BURST\_CREAD\_UNPACKED4 · 285  
P2\_BURST\_CREAD\_UNPACKED8 · 287  
BURST\_CSTART · 27  
BURST\_INIT · 28  
P2\_BURST\_INIT · 289  
BURST\_READ · 30  
P2\_BURST\_READ · 294  
P2\_BURST\_READ\_INDEX · 292  
BURST\_READ\_PACKED · 32  
P2\_BURST\_READ\_UNPACKED1 · 296  
P2\_BURST\_READ\_UNPACKED2 · 298  
P2\_BURST\_READ\_UNPACKED4 · 300  
P2\_BURST\_READ\_UNPACKED8 · 302  
P2\_BURST\_RESET · 304  
BURST\_START · 34  
P2\_BURST\_START · 306  
BURST\_STATUS · 36  
P2\_BURST\_STATUS · 307  
P2\_BURST\_STOP · 309

#### C

CAN\_MSG · 179  
CHANGED\_DATA · 194  
CHECKLED · 4  
CHECK\_ACCESS · 195  
P2\_CHECK\_LED · 222  
CHECK\_SHIFT\_REG · 204  
CNT\_CLEAR · 87  
CNT\_ENABLE · 88  
CNT\_LATCH · 89  
CNT\_READ16 · 90  
CNT\_READ32 · 91  
CNT\_READLATCH16 · 92  
CNT\_READLATCH32 · 93  
CNT\_SETMODE · 94

CO4\_CLEARENABLE · 95  
CO4\_GETSTATUS · 96  
CO4\_LATCHENABLE · 98  
CO4\_READ · 99  
CO4\_READLATCH · 100  
CO4\_RESETSTATUS · 101  
CO4\_SETMODE · 103  
CO4\_SET\_LATCHMODE · 102  
COMP\_DIGIN\_WORD · 139  
COMP\_DIGIN\_WORD\_DIFF · 140  
COMP\_FIFO\_READ · 141  
COMP\_FIFO\_SELECT · 142  
COMP\_READ · 143  
COMP\_RESET · 144  
COMP\_SET · 145  
CPU\_DIGIN (T11) · 223  
CPU\_DIGIN (T9, T10) · 5  
CPU\_DIGOUT · 224  
CPU\_DIG\_IO\_CONFIG · 225  
CPU\_EVENT\_CONFIG · 226

#### D

DAC · 73  
P2\_DAC · 312  
P2\_DAC4 · 313  
P2\_DAC4\_PACKED · 314  
P2\_DAC8 · 316  
P2\_DAC8\_PACKED · 317  
DIGIN\_LONG\_F · 114  
DIGIN\_WORD1 · 115  
DIGIN\_WORD2 · 116  
DIGOUT · 117  
DIGOUT\_BITS\_F · 119  
DIGOUT\_F · 120  
DIGOUT\_LONG\_F · 121  
DIGOUT\_WORD1 · 122  
DIGOUT\_WORD2 · 123  
DIGPROG1 · 124  
DIGPROG2 · 125  
DIG\_LATCH · 105  
DIG\_READLATCH1 · 107  
DIG\_READLATCH2 · 108  
DIG\_WRITELATCH1 · 109  
DIG\_WRITELATCH2 · 111  
DIG\_WRITELATCH32 · 113

#### E

EN\_INTERRUPT · 181  
EN\_RECEIVE · 182  
EN\_TRANSMIT · 183  
P2\_EVENT2\_CONFIG · 229  
EVENTENABLE · 6  
P2\_EVENT\_CONFIG · 228  
P2\_EVENT\_ENABLE · 227

P2\_EVENT\_READ · 231  
EXTLCH\_ENABLE · 126

## F

FG\_CONTROL · 74  
FG\_DEF · 76  
FG\_DELAY · 77  
FG\_MODE · 78  
FG\_READ\_INDEX · 80  
FG\_STATUS · 81  
FG\_WRITE · 82

## G

GET\_CAN\_REG · 184  
GET\_DIGOUT\_LONG · 127  
GET\_DIGOUT\_WORD1 · 128  
GET\_DIGOUT\_WORD2 · 129  
GET\_PRO\_BYTE · 196  
GET\_READ\_BUFFER · 197  
GET\_RS · 205

## I

INIT\_CAN · 185  
INIT\_SLAVE · 198

## L

LS\_DIGPROG · 215  
LS\_DIG\_IO · 217  
LS\_DIO\_INIT · 213  
LS\_WATCHDOG\_INIT · 219

## M

MEDIA\_RD\_BLK\_F · 159  
MEDIA\_RD\_BLK\_L · 155  
MEDIA\_RD\_FILEINFO · 161  
MEDIA\_WR\_BLK\_F · 153  
MEDIA\_WR\_BLK\_L · 149

## P

P2\_SEQ\_INIT · 249  
P2\_SEQ\_READ · 252  
P2\_SEQ\_READ24 · 253  
P2\_SEQ\_READ\_PACKED · 254  
P2\_SEQ\_START · 255  
P2\_SEQ\_WAIT · 256  
P2\_SET\_MUX · 257  
PT100\_DIG\_TO\_R · 165  
PT100\_DIG\_TO\_TEMP · 164  
PWM\_ENABLE · 130  
PWM\_OUT · 131  
PWM\_SET · 132

## R

READADC · 38  
READADCF · 40  
READADCF\_32 · 45  
READADCF\_SCONV · 46

READADCF\_SCONV\_32 · 47  
READADC\_SCONV · 39  
P2\_READ\_ADC · 244  
P2\_READ\_ADC24 · 245  
P2\_READ\_ADCF · 267  
P2\_READ\_ADCF32 · 275  
P2\_READ\_ADCF4 · 269  
READ\_ADCF4 · 41  
P2\_READ\_ADCF4\_24B · 270  
P2\_READ\_ADCF4\_PACKED · 273  
READ\_ADCF4\_PACKED · 43  
P2\_READ\_ADCF8 · 271  
READ\_ADCF8 · 42  
P2\_READ\_ADCF8\_24B · 272  
P2\_READ\_ADCF8\_PACKED · 274  
READ\_ADCF8\_PACKED · 44  
P2\_READ\_ADCF\_24 · 268  
P2\_READ\_ADCF\_SCONV · 276  
P2\_READ\_ADCF\_SCONV24 · 277  
P2\_READ\_ADCF\_SCONV32 · 278  
P2\_READ\_ADC\_SCONV · 246  
P2\_READ\_ADC\_SCONV24 · 247  
READ\_FIFO · 206  
READ\_MSG · 186  
REQUEST\_ACCESS · 200  
REQUEST\_RELEASE\_ACCESS · 201  
RESETWATCHDOGTIMER · 7  
RS485\_SEND · 210  
RS\_INIT · 207  
RS\_RESET · 209  
RTC\_GET · 148  
RTC\_SET · 147

## S

SEQ\_MODE · 52  
SEQ\_READ · 54  
SEQ\_READ32 · 60  
SEQ\_READ\_ONE · 55  
SEQ\_READ\_PACKED · 58  
SEQ\_READ\_TWO · 56  
SEQ\_SELECT · 61  
SEQ\_SET\_DELAY · 62  
SEQ\_STATUS · 64  
SETLED · 8  
P2\_SET\_AVERAGE\_FILTER · 310  
SET\_CAN\_BAUDRATE · 188  
SET\_CAN\_REG · 192  
SET\_GAIN · 49  
P2\_SET\_LED · 232  
SET\_MUX · 50  
SET\_PRO\_BYTE · 202  
SET\_RS · 211  
SET\_WRITE\_BUFFER · 203  
P2\_SE\_DIFF · 248  
SE\_DIFF · 48  
SH\_SETMODE · 65  
SSI\_MODE · 133  
SSI\_READ · 134  
SSI\_SET\_BITS · 135

SSI\_SET\_CLOCK · 136  
SSI\_START · 137  
SSI\_STATUS · 138  
STARTWATCHDOG · 10  
START\_CONV · 66  
P2\_START\_CONV · 258  
P2\_START\_CONVF · 279  
START\_CONVF · 67  
P2\_START\_DAC · 319  
START\_DAC · 84  
STOPWATCHDOG · 11  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16  
P2\_SYNC\_ALL · 233  
P2\_SYNC\_ENABLE · 235  
SYNC\_MODE · 68  
P2\_SYNC\_STAT · 237

### T

TCJ\_DIG\_TO\_TEMP · 168  
TCK\_DIG\_TO\_TEMP · 169  
TC\_READ\_B · 170  
TC\_READ\_E · 171  
TC\_READ\_J · 172  
TC\_READ\_K · 173  
TC\_READ\_N · 174  
TC\_READ\_R · 175  
TC\_READ\_S · 176  
TC\_READ\_T · 177  
TC\_SELECT · 166  
TC\_SET\_RATE · 178  
TRANSMIT · 193

### W

P2\_WAIT\_EOC · 259  
WAIT\_EOC · 70  
P2\_WAIT\_EOCF · 280  
WAIT\_EOCF · 71  
WRITEDAC · 85  
P2\_WRITE\_DAC · 320  
P2\_WRITE\_DAC32 · 325  
P2\_WRITE\_DAC4 · 321  
P2\_WRITE\_DAC4\_PACKED · 322  
P2\_WRITE\_DAC8 · 323  
P2\_WRITE\_DAC8\_PACKED · 324  
WRITE\_FIFO · 212





## A.2 Instruction List sorted by Module Types

You find the instruction lists of the modules on these pages:

From module	To module	Page
<a href="#">Aln-16/14-C Rev. A</a>	<a href="#">Aln-32/12 Rev. B</a>	A-5
<a href="#">Aln-32/14 Rev. A</a>	<a href="#">Aln-32/18 Rev. E</a>	A-6
<a href="#">Aln-8/12 Rev. A</a>	<a href="#">Aln-8/16 Rev. C</a>	A-7
<a href="#">Aln-8/18 Rev. E</a>	<a href="#">Aln-F-4/16 Rev. A</a>	A-8
<a href="#">Aln-F-8/12 Rev. A</a>	<a href="#">Aln-F-8/16 Rev. A</a>	A-9
<a href="#">Aln-F-8/18 Rev. E</a>	<a href="#">AOut-4/16 Rev. E</a>	A-10
<a href="#">AOut-8/16 Rev. A</a>	<a href="#">CNT-8/32(-I)</a>	A-11
<a href="#">CNT-PW4(-I)</a>	<a href="#">CPU-T11</a>	A-12
<a href="#">CPU-T9</a>	<a href="#">LS2 Rev. A</a>	A-13
<a href="#">OPT-16 Rev. A</a>	<a href="#">RS422-2, RS422-4</a>	A-14
<a href="#">RS485-2, RS485-4</a>	<a href="#">(LP)SH-8(-FI)</a>	A-15

### Aln-16/14-C Rev. A

**A:** [ADC](#) · 18  
**C:** [CHECKLED](#) · 4  
**R:** [READADC](#) · 38  
[READADC\\_SCONV](#) · 39  
**S:** [SEQ\\_MODE](#) · 52  
[SEQ\\_READ](#) · 54  
[SEQ\\_READ32](#) · 60  
[SEQ\\_READ\\_ONE](#) · 55  
[SEQ\\_READ\\_PACKED](#) · 58  
[SEQ\\_READ\\_TWO](#) · 56  
[SEQ\\_SELECT](#) · 61  
[SEQ\\_SET\\_DELAY](#) · 62  
[SEQ\\_STATUS](#) · 64  
[SETLED](#) · 8  
[SET\\_MUX](#) · 50  
[START\\_CONV](#) · 66  
[SYNCALL](#) · 12  
[SYNCENABLE](#) · 14  
[SYNCSTAT](#) · 16  
**W:** [WAIT\\_EOC](#) · 70

### Aln-32/12 Rev. A

**A:** [ADC](#) · 18  
**C:** [CHECKLED](#) · 4  
**R:** [READADC](#) · 38  
**S:** [SETLED](#) · 8  
[SET\\_MUX](#) · 50  
[SE\\_DIFF](#) · 48  
[START\\_CONV](#) · 66  
[SYNCALL](#) · 12  
[SYNCENABLE](#) · 14  
[SYNCSTAT](#) · 16

### Aln-32/12 Rev. B

**A:** [ADC](#) · 18  
**C:** [CHECKLED](#) · 4  
**R:** [READADC](#) · 38  
[READADC\\_SCONV](#) · 39  
**S:** [SETLED](#) · 8  
[SET\\_MUX](#) · 50  
[SE\\_DIFF](#) · 48  
[START\\_CONV](#) · 66  
[SYNCALL](#) · 12  
[SYNCENABLE](#) · 14  
[SYNCSTAT](#) · 16  
**W:** [WAIT\\_EOC](#) · 70

**Aln-32/14 Rev. A**

**A:** ADC · 18  
**C:** CHECKLED · 4  
**R:** READADC · 38  
    READADC\_SCONV · 39  
**S:** SEQ\_MODE · 52  
    SEQ\_READ · 54  
    SEQ\_READ32 · 60  
    SEQ\_READ\_ONE · 55  
    SEQ\_READ\_PACKED · 58  
    SEQ\_READ\_TWO · 56  
    SEQ\_SELECT · 61  
    SEQ\_SET\_DELAY · 62  
    SEQ\_STATUS · 64  
    SETLED · 8  
    SET\_MUX · 50  
    SE\_DIFF · 48  
    START\_CONV · 66  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
**W:** WAIT\_EOC · 70

**Aln-32/16 Rev. B**

**A:** ADC · 18  
**C:** CHECKLED · 4  
**R:** READADC · 38  
    READADC\_SCONV · 39  
**S:** SETLED · 8  
    SET\_MUX · 50  
    SE\_DIFF · 48  
    START\_CONV · 66  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
**W:** WAIT\_EOC · 70

**Aln-32/16 Rev. C**

**A:** ADC · 18  
**C:** CHECKLED · 4  
**R:** READADC · 38  
    READADC\_SCONV · 39  
**S:** SEQ\_MODE · 52  
    SEQ\_READ · 54  
    SEQ\_READ32 · 60  
    SEQ\_READ\_ONE · 55  
    SEQ\_READ\_PACKED · 58  
    SEQ\_READ\_TWO · 56  
    SEQ\_SELECT · 61  
    SEQ\_SET\_DELAY · 62  
    SEQ\_STATUS · 64  
    SETLED · 8  
    SET\_MUX · 50  
    SE\_DIFF · 48  
    START\_CONV · 66  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
**W:** WAIT\_EOC · 70

**Aln-32/18 Rev. E**

**A:** P2\_ADC · 239  
    P2\_ADC24 · 240  
    P2\_ADC\_READ\_LIMIT · 241  
    P2\_ADC\_SET\_LIMIT · 243  
**C:** P2\_CHECK\_LED · 222  
**R:** P2\_READ\_ADC · 244  
    P2\_READ\_ADC24 · 245  
    P2\_READ\_ADC\_SCONV · 246  
    P2\_READ\_ADC\_SCONV24 · 247  
**S:** P2\_SEQ\_INIT · 249  
    P2\_SEQ\_READ · 252  
    P2\_SEQ\_READ24 · 253  
    P2\_SEQ\_READ\_PACKED · 254  
    P2\_SEQ\_START · 255  
    P2\_SEQ\_WAIT · 256  
    P2\_SET\_LED · 232  
    P2\_SET\_MUX · 257  
    P2\_SE\_DIFF · 248  
    P2\_START\_CONV · 258  
    P2\_SYNC\_ALL · 233  
**W:** P2\_WAIT\_EOC · 259

## **Aln-8/12 Rev. A**

**A:** ADC · 18  
**C:** CHECKLED · 4  
**R:** READADC · 38  
**S:** SETLED · 8  
    SET\_MUX · 50  
    START\_CONV · 66  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
**W:** WAIT\_EOC · 70

## **Aln-8/12 Rev. B**

**A:** ADC · 18  
**C:** CHECKLED · 4  
**R:** READADC · 38  
    READADC\_SCONV · 39  
**S:** SETLED · 8  
    SET\_MUX · 50  
    START\_CONV · 66  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
**W:** WAIT\_EOC · 70

## **Aln-8/14 Rev. A**

**A:** ADC · 18  
**C:** CHECKLED · 4  
**R:** READADC · 38  
    READADC\_SCONV · 39  
**S:** SEQ\_MODE · 52  
    SEQ\_READ · 54  
    SEQ\_READ32 · 60  
    SEQ\_READ\_ONE · 55  
    SEQ\_READ\_PACKED · 58  
    SEQ\_READ\_TWO · 56  
    SEQ\_SELECT · 61  
    SEQ\_SET\_DELAY · 62  
    SEQ\_STATUS · 64  
    SETLED · 8  
    SET\_MUX · 50  
    START\_CONV · 66  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
**W:** WAIT\_EOC · 70

## **Aln-8/16 Rev. A**

**A:** ADC16 · 20  
**C:** CHECKLED · 4  
**R:** READADC · 38  
    READADC\_SCONV · 39  
**S:** SETLED · 8  
    SET\_MUX · 50  
    START\_CONV · 66  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
**W:** WAIT\_EOC · 70

## **Aln-8/16 Rev. B**

**A:** ADC · 18  
**C:** CHECKLED · 4  
**R:** READADC · 38  
    READADC\_SCONV · 39  
**S:** SETLED · 8  
    SET\_MUX · 50  
    START\_CONV · 66  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
**W:** WAIT\_EOC · 70

## **Aln-8/16 Rev. C**

**A:** ADC · 18  
**C:** CHECKLED · 4  
**R:** READADC · 38  
    READADC\_SCONV · 39  
**S:** SEQ\_MODE · 52  
    SEQ\_READ · 54  
    SEQ\_READ32 · 60  
    SEQ\_READ\_ONE · 55  
    SEQ\_READ\_PACKED · 58  
    SEQ\_READ\_TWO · 56  
    SEQ\_SELECT · 61  
    SEQ\_SET\_DELAY · 62  
    SEQ\_STATUS · 64  
    SETLED · 8  
    SET\_MUX · 50  
    START\_CONV · 66  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
**W:** WAIT\_EOC · 70

**Aln-8/18 Rev. E**

- A:** P2\_ADC · 239  
P2\_ADC24 · 240  
P2\_ADC\_READ\_LIMIT · 241  
P2\_ADC\_SET\_LIMIT · 243
- C:** P2\_CHECK\_LED · 222
- R:** P2\_READ\_ADC · 244  
P2\_READ\_ADC24 · 245  
P2\_READ\_ADC\_SCONV · 246  
P2\_READ\_ADC\_SCONV24 · 247
- S:** P2\_SEQ\_INIT · 249  
P2\_SEQ\_READ · 252  
P2\_SEQ\_READ24 · 253  
P2\_SEQ\_READ\_PACKED · 254  
P2\_SEQ\_START · 255  
P2\_SEQ\_WAIT · 256  
P2\_SET\_LED · 232  
P2\_SET\_MUX · 257  
P2\_START\_CONV · 258  
P2\_SYNC\_ALL · 233
- W:** P2\_WAIT\_EOC · 259

**Aln-F-4/12 Rev. A**

- A:** ADCF · 22
- C:** CHECKLED · 4
- R:** READADCF · 40  
READADCF\_32 · 45  
READADCF\_SCONV · 46  
READADCF\_SCONV\_32 · 47  
READ\_ADCF4 · 41  
READ\_ADCF4\_PACKED · 43
- S:** SETLED · 8  
START\_CONVF · 67  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16
- W:** WAIT\_EOCF · 71

**Aln-F-4/14 Rev. B**

- A:** ADCF · 22
- B:** BURST\_ABORT · 23  
BURST\_CREAD · 25  
BURST\_CSTART · 27  
BURST\_INIT · 28  
BURST\_READ · 30  
BURST\_READ\_PACKED · 32  
BURST\_STATUS · 36  
vBURST\_START · 34
- C:** CHECKLED · 4
- R:** READADCF · 40  
READADCF\_32 · 45  
READADCF\_SCONV · 46  
READADCF\_SCONV\_32 · 47  
READ\_ADCF4 · 41  
READ\_ADCF4\_PACKED · 43
- S:** SETLED · 8  
SET\_GAIN · 49  
START\_CONVF · 67  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16  
SYNC\_MODE · 68
- W:** WAIT\_EOCF · 71

**Aln-F-4/16 Rev. A**

- A:** ADCF · 22
- C:** CHECKLED · 4
- R:** READADCF · 40  
READADCF\_32 · 45  
READADCF\_SCONV · 46  
READADCF\_SCONV\_32 · 47  
READ\_ADCF4 · 41  
READ\_ADCF4\_PACKED · 43
- S:** SETLED · 8  
START\_CONVF · 67  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16
- W:** WAIT\_EOCF · 71

## **Aln-F-8/12 Rev. A**

**A:** ADCF · 22  
**C:** CHECKLED · 4  
**R:** READADCF · 40  
    READADCF\_32 · 45  
    READADCF\_SCONV · 46  
    READADCF\_SCONV\_32 · 47  
    READ\_ADCF4 · 41  
    READ\_ADCF4\_PACKED · 43  
    READ\_ADCF8 · 42  
    READ\_ADCF8\_PACKED · 44  
**S:** SETTLED · 8  
    START\_CONVF · 67  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
**W:** WAIT\_EOCF · 71

## **Aln-F-8/14 Rev. B**

**A:** ADCF · 22  
**B:** BURST\_ABORT · 23  
    BURST\_CREAD · 25  
    BURST\_CSTART · 27  
    BURST\_INIT · 28  
    BURST\_READ · 30  
    BURST\_READ\_PACKED · 32  
    BURST\_START · 34  
    BURST\_STATUS · 36  
**C:** CHECKLED · 4  
**R:** READADCF · 40  
    READADCF\_32 · 45  
    READADCF\_SCONV · 46  
    READADCF\_SCONV\_32 · 47  
    READ\_ADCF4 · 41  
    READ\_ADCF4\_PACKED · 43  
    READ\_ADCF8 · 42  
    READ\_ADCF8\_PACKED · 44  
**S:** SETTLED · 8  
    SET\_GAIN · 49  
    START\_CONVF · 67  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
    SYNC\_MODE · 68  
**W:** WAIT\_EOCF · 71

## **Aln-F-8/14 Rev. E**

**A:** P2\_ADCF\_READ\_LIMIT · 265  
    P2\_ADCF\_SET\_LIMIT · 266  
**B:** P2\_BURST\_CREAD\_UNPACKED1 · 281  
    P2\_BURST\_CREAD\_UNPACKED2 · 283  
    P2\_BURST\_CREAD\_UNPACKED4 · 285  
    P2\_BURST\_CREAD\_UNPACKED8 · 287  
    P2\_BURST\_INIT · 289  
    P2\_BURST\_READ · 294  
    P2\_BURST\_READ\_INDEX · 292  
    P2\_BURST\_READ\_UNPACKED1 · 296  
    P2\_BURST\_READ\_UNPACKED2 · 298  
    P2\_BURST\_READ\_UNPACKED4 · 300  
    P2\_BURST\_READ\_UNPACKED8 · 302  
    P2\_BURST\_RESET · 304  
    P2\_BURST\_START · 306  
    P2\_BURST\_STATUS · 307  
    P2\_BURST\_STOP · 309  
**C:** P2\_CHECK\_LED · 222  
**E:** P2\_EVENT2\_CONFIG · 229  
    P2\_EVENT\_CONFIG · 228  
    P2\_EVENT\_ENABLE · 227  
    P2\_EVENT\_READ · 231  
**R:** P2\_READ\_ADCF · 267  
    P2\_READ\_ADCF4 · 269  
    P2\_READ\_ADCF4\_PACKED · 273  
    P2\_READ\_ADCF8 · 271  
    P2\_READ\_ADCF8\_PACKED · 274  
    P2\_READ\_ADCF\_32 · 275  
**S:** P2\_SET\_AVERAGE\_FILTER · 310  
    P2\_SET\_LED · 232  
    P2\_SYNC\_ALL · 233  
    P2\_SYNC\_ENABLE · 235  
    P2\_SYNC\_STAT · 237

## **Aln-F-8/16 Rev. A**

**A:** ADCF · 22  
**C:** CHECKLED · 4  
**R:** READADCF · 40  
    READADCF\_32 · 45  
    READADCF\_SCONV · 46  
    READADCF\_SCONV\_32 · 47  
    READ\_ADCF4 · 41  
    READ\_ADCF4\_PACKED · 43  
    READ\_ADCF8 · 42  
    READ\_ADCF8\_PACKED · 44  
**S:** SETTLED · 8  
    START\_CONVF · 67  
    SYNCALL · 12  
    SYNCENABLE · 14  
    SYNCSTAT · 16  
**W:** WAIT\_EOCF · 71

**Aln-F-8/18 Rev. E**

- A:** P2\_ADCF · 260  
P2\_ADCF24 · 261  
P2\_ADCF\_MODE · 262  
P2\_ADCF\_READ\_LIMIT · 265  
P2\_ADCF\_SET\_LIMIT · 266
- C:** P2\_CHECK\_LED · 222
- E:** P2\_EVENT2\_CONFIG · 229  
P2\_EVENT\_CONFIG · 228  
P2\_EVENT\_ENABLE · 227  
P2\_EVENT\_READ · 231
- R:** P2\_READ\_ADCF · 267  
P2\_READ\_ADCF4 · 269  
P2\_READ\_ADCF4\_24B · 270  
P2\_READ\_ADCF4\_PACKED · 273  
P2\_READ\_ADCF8 · 271  
P2\_READ\_ADCF8\_24B · 272  
P2\_READ\_ADCF8\_PACKED · 274  
P2\_READ\_ADCF\_24 · 268  
P2\_READ\_ADCF\_32 · 275  
P2\_READ\_ADCF\_SCONV · 276  
P2\_READ\_ADCF\_SCONV24 · 277  
P2\_READ\_ADCF\_SCONV32 · 278
- S:** P2\_SET\_LED · 232  
P2\_START\_CONVF · 279  
P2\_SYNC\_ALL · 233  
P2\_SYNC\_ENABLE · 235  
P2\_SYNC\_STAT · 237
- W:** P2\_WAIT\_EOCF · 280

**AO-16/8-12**

- R:** READADC · 38  
READADC\_SCONV · 39
- S:** SET\_MUX · 50  
START\_CONV · 66
- W:** WAIT\_EOC · 70

**AOut-16/8-12**

- C:** CHECKLED · 4
- D:** DAC · 73
- S:** SETLED · 8  
START\_DAC · 84  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16
- W:** WRITEDAC · 85

**AOut-4/16 Rev. A**

- C:** CHECKLED · 4
- D:** DAC · 73
- S:** SETLED · 8  
START\_DAC · 84  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16
- W:** WRITEDAC · 85

**AOut-4/16 Rev. B**

- C:** CHECKLED · 4
- D:** DAC · 73
- S:** SETLED · 8  
START\_DAC · 84  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16
- W:** WRITEDAC · 85

**AOut-4/16 Rev. C**

- C:** CHECKLED · 4
- D:** DAC · 73
- F:** FG\_CONTROL (AOut-4/16-M2 only) · 74  
FG\_DEF (AOut-4/16-M2 only) · 76  
FG\_DELAY (AOut-4/16-M2 only) · 77  
FG\_Mode (AOut-4/16-M2 only) · 78  
FG\_READ\_INDEX (AOut-4/16-M2 only) · 80  
FG\_STATUS (AOut-4/16-M2 only) · 81  
FG\_WRITE (AOut-4/16-M2 only) · 82
- S:** SETLED · 8  
START\_DAC · 84  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16
- W:** WRITEDAC · 85

**AOut-4/16 Rev. E**

- C:** P2\_CHECK\_LED · 222
- D:** P2\_DAC · 312  
P2\_DAC4 · 313  
P2\_DAC4\_PACKED · 314
- E:** P2\_EVENT\_CONFIG · 228  
P2\_EVENT\_ENABLE · 227  
P2\_EVENT\_READ · 231
- S:** P2\_SET\_LED · 232  
P2\_START\_DAC · 319  
P2\_SYNC\_ALL · 233  
P2\_SYNC\_ENABLE · 235  
P2\_SYNC\_STAT · 237
- W:** P2\_WRITE\_DAC · 320  
P2\_WRITE\_DAC32 · 325  
P2\_WRITE\_DAC4 · 321  
P2\_WRITE\_DAC4\_PACKED · 322

## **AOut-8/16 Rev. A**

**C:** CHECKLED · 4  
**D:** DAC · 73  
**S:** SETLED · 8  
START\_DAC · 84  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16  
**W:** WRITEDAC · 85

## **AOut-8/16 Rev. B**

**C:** CHECKLED · 4  
**D:** DAC · 73  
**S:** SETLED · 8  
START\_DAC · 84  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16  
**W:** WRITEDAC · 85

## **AOut-8/16 Rev. C**

**C:** CHECKLED · 4  
**D:** DAC · 73  
**S:** SETLED · 8  
START\_DAC · 84  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16  
**W:** WRITEDAC · 85

## **AOut-8/16 Rev. E**

**C:** P2\_CHECK\_LED · 222  
**D:** P2\_DAC · 312  
P2\_DAC4 · 313  
P2\_DAC4\_PACKED · 314  
P2\_DAC8 · 316  
P2\_DAC8\_PACKED · 317  
**E:** P2\_EVENT\_CONFIG · 228  
P2\_EVENT\_ENABLE · 227  
P2\_EVENT\_READ · 231  
**S:** P2\_SET\_LED · 232  
P2\_START\_DAC · 319  
P2\_SYNC\_ALL · 233  
P2\_SYNC\_ENABLE · 235  
P2\_SYNC\_STAT · 237  
**W:** P2\_WRITE\_DAC · 320  
P2\_WRITE\_DAC32 · 325  
P2\_WRITE\_DAC4 · 321  
P2\_WRITE\_DAC4\_PACKED · 322  
P2\_WRITE\_DAC8 · 323  
P2\_WRITE\_DAC8\_PACKED · 324

## **CAN-1, CAN-2**

**C:** CAN\_MSG · 179  
CHECKLED · 4  
**E:** EN\_INTERRUPT · 181  
EN\_RECEIVE · 182  
EN\_TRANSMIT · 183  
**G:** GET\_CAN\_REG · 184  
**I:** INIT\_CAN · 185  
**R:** READ\_MSG · 186  
**S:** SETLED · 8  
SET\_CAN\_BAUDRATE · 188  
SET\_CAN\_REG · 192  
**T:** TRANSMIT · 193

## **CNT-16/16(-I)**

**C:** CHECKLED · 4  
CNT\_CLEAR · 87  
CNT\_ENABLE · 88  
CNT\_LATCH · 89  
CNT\_READ16 · 90  
CNT\_READLATCH16 · 92  
**E:** EVENTENABLE · 6  
**S:** SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

## **CNT-16/32(-I)**

**C:** CHECKLED · 4  
CNT\_CLEAR · 87  
CNT\_ENABLE · 88  
CNT\_LATCH · 89  
CO4\_READ · 99  
CO4\_READLATCH · 100  
**E:** EVENTENABLE · 6  
**S:** SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

## **CNT-8/32(-I)**

**C:** CHECKLED · 4  
CNT\_CLEAR · 87  
CNT\_ENABLE · 88  
CNT\_LATCH · 89  
CNT\_READ32 · 91  
CNT\_READLATCH32 · 93  
**E:** EVENTENABLE · 6  
**S:** SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

## **CNT-PW4(-I)**

- C:** CHECKLED · 4  
CNT\_CLEAR · 87  
CNT\_ENABLE · 88  
CNT\_LATCH · 89  
CNT\_READ32 · 91  
CNT\_READLATCH32 · 93
- E:** EVENTENABLE · 6
- S:** SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

## **CNT-VR2PW2**

- C:** CNT\_CLEAR · 87  
CNT\_ENABLE · 88  
CNT\_LATCH · 89  
CNT\_READ32 · 91  
CNT\_READLATCH32 · 93  
CNT\_SETMODE · 94

## **CNT-VR4L(-I)**

- C:** CHECKLED · 4  
CNT\_CLEAR · 87  
CNT\_ENABLE · 88  
CNT\_LATCH · 89  
CNT\_READ32 · 91  
CNT\_READLATCH32 · 93  
CNT\_SETMODE · 94
- E:** EVENTENABLE · 6  
EXTLCH\_ENABLE · 126
- S:** SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

## **CNT-VR4(-I)**

- C:** CHECKLED · 4  
CNT\_CLEAR · 87  
CNT\_ENABLE · 88  
CNT\_LATCH · 89  
CNT\_READ32 · 91  
CNT\_READLATCH32 · 93  
CNT\_SETMODE · 94
- E:** EVENTENABLE · 6
- S:** SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

## **CO4**

- C:** CHECKLED · 4  
CNT\_CLEAR · 87  
CNT\_ENABLE · 88  
CNT\_LATCH · 89  
CO4\_CLEARENABLE · 95  
CO4\_GETSTATUS · 96  
CO4\_LATCHENABLE · 98  
CO4\_READ · 99  
CO4\_READLATCH · 100  
CO4\_RESETSTATUS · 101  
CO4\_SETMODE · 103  
CO4\_SET\_LATCHMODE · 102
- E:** EVENTENABLE · 6
- S:** SETLED · 8  
SSI\_MODE · 133  
SSI\_READ · 134  
SSI\_SET\_BITS · 135  
SSI\_SET\_CLOCK · 136  
SSI\_START · 137  
SSI\_STATUS · 138  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

## **COMP16 Rev. A**

- C:** COMP\_DIGIN\_WORD · 139  
COMP\_DIGIN\_WORD\_DIFF · 140  
COMP\_FIFO\_READ · 141  
COMP\_FIFO\_SELECT · 142  
COMP\_READ · 143  
COMP\_RESET · 144  
COMP\_SET · 145

## **CPU-T10**

- C:** CHECKLED · 4  
CPU\_DIGIN · 5
- R:** RESETWATCHDOGTIMER · 7
- S:** SETLED · 8  
STARTWATCHDOG · 10  
STOPWATCHDOG · 11

## **CPU-T11**

- C:** CPU\_DIGIN · 223  
CPU\_DIGOUT · 224  
CPU\_DIG\_IO\_CONFIG · 225  
CPU\_EVENT\_CONFIG · 226  
P2\_CHECK\_LED · 222
- R:** RESETWATCHDOGTIMER · 7
- S:** P2\_SET\_LED · 232  
STARTWATCHDOG · 10  
STOPWATCHDOG · 11



### CPU-T9

- C: CHECKLED · 4  
CPU\_DIGIN (module option only) · 5
- R: RESETWATCHDOGTIMER · 7
- S: SETLED · 8  
STARTWATCHDOG · 10  
STOPWATCHDOG · 11

### DIO-32

- C: CHECKLED · 4
- D: DIGIN\_WORD1 · 115  
DIGIN\_WORD2 · 116  
DIGOUT · 117  
DIGOUT\_WORD1 · 122  
DIGOUT\_WORD2 · 123  
DIGPROG1 · 124  
DIGPROG2 · 125  
DIG\_LATCH · 105  
DIG\_READLATCH1 · 107  
DIG\_READLATCH2 · 108  
DIG\_WRITELATCH1 · 109  
DIG\_WRITELATCH2 · 111  
DIG\_WRITELATCH32 · 113
- E: EVENTENABLE · 6
- G: GET\_DIGOUT\_LONG · 127  
GET\_DIGOUT\_WORD1 · 128  
GET\_DIGOUT\_WORD2 · 129
- S: SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

### DIO-32 Rev. B

- C: CHECKLED · 4
- D: DIGIN\_LONG\_F · 114  
DIGIN\_WORD1 · 115  
DIGIN\_WORD2 · 116  
DIGOUT · 117  
DIGOUT\_BITS\_F · 119  
DIGOUT\_F · 120  
DIGOUT\_LONG\_F · 121  
DIGOUT\_WORD1 · 122  
DIGOUT\_WORD2 · 123  
DIGPROG1 · 124  
DIGPROG2 · 125  
DIG\_LATCH · 105  
DIG\_READLATCH1 · 107  
DIG\_READLATCH2 · 108  
DIG\_WRITELATCH1 · 109  
DIG\_WRITELATCH2 · 111  
DIG\_WRITELATCH32 · 113
- E: EVENTENABLE · 6
- G: GET\_DIGOUT\_WORD1 · 128  
GET\_DIGOUT\_WORD2 · 129
- S: SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

### INTER-SL

- C: CHECKLED · 4
- S: SETLED · 8

### Inter-SL

- C: CHANGED\_DATA · 194  
CHECK\_ACCESS · 195
- G: GET\_PRO\_BYTE · 196  
GET\_READ\_BUFFER · 197
- I: INIT\_SLAVE · 198
- R: REQUEST\_ACCESS · 200  
REQUEST\_RELEASE\_ACCESS · 201
- S: SET\_PRO\_BYTE · 202  
SET\_WRITE\_BUFFER · 203

### LS-2 Rev. A

- L: LS\_DIGPROG · 215  
LS\_DIG\_IO · 217  
LS\_DIO\_INIT · 213  
LS\_WATCHDOG\_INIT · 219

### LS2 Rev. A

- C: CHECKLED · 4
- S: SETLED · 8

**OPT-16 Rev. A**

C: CHECKLED · 4  
D: DIGIN\_WORD1 · 115  
DIG\_LATCH · 105  
DIG\_READLATCH1 · 107  
E: EVENTENABLE · 6  
S: SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

**OPT-16 Rev. B**

C: CHECKLED · 4  
D: DIGIN\_WORD1 · 115  
DIG\_LATCH · 105  
DIG\_READLATCH1 · 107  
E: EVENTENABLE · 6  
S: SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

**Profi-SL**

C: CHANGED\_DATA · 194  
CHECKLED · 4  
CHECK\_ACCESS · 195  
G: GET\_PRO\_BYTE · 196  
GET\_READ\_BUFFER · 197  
I: INIT\_SLAVE · 198  
R: REQUEST\_ACCESS · 200  
REQUEST\_RELEASE\_ACCESS · 201  
S: SETLED · 8  
SET\_PRO\_BYTE · 202  
SET\_WRITE\_BUFFER · 203

**PT100-4, PT100-8**

C: CHECKLED · 4  
P: PT100\_DIG\_TO\_R · 165  
PT100\_DIG\_TO\_TEMP · 164  
S: SETLED · 8  
T: TC\_SELECT · 166

**PWM-4(-I)**

C: CHECKLED · 4  
E: EVENTENABLE · 6  
P: PWM\_ENABLE · 130  
PWM\_OUT · 131  
PWM\_SET · 132  
S: SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

**REL-16 Rev. A**

C: CHECKLED · 4  
D: DIGOUT · 117  
DIGOUT\_WORD1 · 122  
DIG\_LATCH · 105  
DIG\_WRITELATCH1 · 109  
E: EVENTENABLE · 6  
G: GET\_DIGOUT\_WORD1 · 128  
S: SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

**REL-16 Rev. B**

C: CHECKLED · 4  
D: DIGOUT · 117  
DIGOUT\_WORD1 · 122  
DIG\_LATCH · 105  
DIG\_WRITELATCH1 · 109  
E: EVENTENABLE · 6  
G: GET\_DIGOUT\_WORD1 · 128  
S: SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

**RS232-2, RS232-4**

C: CHECKLED · 4  
CHECK\_SHIFT\_REG · 204  
G: GET\_RS · 205  
R: READ\_FIFO · 206  
RS\_INIT · 207  
RS\_RESET · 209  
S: SETLED · 8  
SET\_RS · 211  
W: WRITE\_FIFO · 212

**RS422-2, RS422-4**

C: CHECK\_SHIFT\_REG · 204  
G: GET\_RS · 205  
R: READ\_FIFO · 206  
RS\_INIT · 207  
RS\_RESET · 209  
S: SET\_RS · 211  
W: WRITE\_FIFO · 212

## RS485-2, RS485-4

**C:** CHECKLED · 4  
CHECK\_SHIFT\_REG · 204  
**G:** GET\_RS · 205  
**R:** READ\_FIFO · 206  
RS485\_SEND · 210  
RS\_INIT · 207  
RS\_RESET · 209  
**S:** SETLED · 8  
SET\_RS · 211  
**W:** WRITE\_FIFO · 212

## Storage Rev. A

**C:** CHECKLED · 4  
**M:** MEDIA\_RD\_BLK\_F · 159  
MEDIA\_RD\_BLK\_L · 155  
MEDIA\_RD\_FILEINFO · 161  
MEDIA\_WR\_BLK\_F · 153  
MEDIA\_WR\_BLK\_L · 149  
**R:** RTC\_GET · 148  
RTC\_SET · 147  
**S:** SETLED · 8

## TC-16

**C:** CHECKLED · 4  
**S:** SETLED · 8  
**T:** TCJ\_DIG\_TO\_TEMP · 168  
TCK\_DIG\_TO\_TEMP · 169  
TC\_SELECT · 166

## TC-4

**C:** CHECKLED · 4  
**S:** SETLED · 8  
**T:** TCJ\_DIG\_TO\_TEMP · 168  
TCK\_DIG\_TO\_TEMP · 169  
TC\_SELECT · 166

## TC-8

**C:** CHECKLED · 4  
**S:** SETLED · 8  
**T:** TCJ\_DIG\_TO\_TEMP · 168  
TCK\_DIG\_TO\_TEMP · 169  
TC\_SELECT · 166

## TC-8-ISO

**T:** TC\_READ\_B · 170  
TC\_READ\_E · 171  
TC\_READ\_J · 172  
TC\_READ\_K · 173  
TC\_READ\_N · 174  
TC\_READ\_R · 175  
TC\_READ\_S · 176  
TC\_READ\_T · 177  
TC\_SET\_RATE · 178

## TRA-16 Rev. A

**C:** CHECKLED · 4  
**D:** DIGOUT · 117  
DIGOUT\_WORD1 · 122  
DIG\_LATCH · 105  
DIG\_WRITELATCH1 · 109  
**E:** EVENTENABLE · 6  
**G:** GET\_DIGOUT\_WORD1 · 128  
**S:** SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

## TRA-16 Rev. B

**C:** CHECKLED · 4  
**D:** DIGOUT · 117  
DIGOUT\_BITS\_F · 119  
DIGOUT\_F · 120  
DIGOUT\_WORD1 · 122  
DIG\_LATCH · 105  
DIG\_WRITELATCH1 · 109  
**E:** EVENTENABLE · 6  
**G:** GET\_DIGOUT\_WORD1 · 128  
**S:** SETLED · 8  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16

## (LP)SH-8(-FI)

**A:** ADC · 18  
ADC16 · 20  
**C:** CHECKLED · 4  
**R:** READADC · 38  
READADC\_SCONV · 39  
**S:** SETLED · 8  
SET\_MUX · 50  
SH\_SETMODE · 65  
START\_CONV · 66  
SYNCALL · 12  
SYNCENABLE · 14  
SYNCSTAT · 16  
**W:** WAIT\_EOC · 70



### A.3 Thematic Instruction List

Die Befehle sind in die folgenden Themengruppen aufgeteilt. Innerhalb der Themengruppen sind die Befehle alphabetisch sortiert.

- Analog Inputs: Seite [A-17](#)
- Analog Outputs: Seite [A-19](#)
- Communication interface: Seite [A-20](#)
- Comparator: Seite [A-21](#)
- Counters: Seite [A-21](#)
- Data Storage: Seite [A-22](#)
- Digital Inputs/Outputs: Seite [A-22](#)
- System: Seite [A-23](#)
- Temperature-Inputs: Seite [A-23](#)

#### Analog Inputs

- [P2\\_ADC](#) runs a complete conversion on an ADC of the specified module. The return value has a resolution of 16 bit.
- [ADC](#) executes a complete measurement process on a 12-bit, 14-bit or 16-bit ADC.
- [ADC16](#) executes a complete measurement on a 16-bit ADC. The information apply only for the module Pro-AIn-8/16 REVA.
- [P2\\_ADC24](#) runs a complete conversion on an ADC of the specified module. The return value has a resolution of 24 bit.
- [P2\\_ADCF](#) executes a complete measurement on a Fast-ADC. The return value has a resolution of 16 bit.
- [ADCF](#) executes a complete measurement on a Fast-ADC.
- [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 -underflow flags of all F-ADCs on the specified module.
- [P2\\_ADCF\\_SET\\_LIMIT](#) sets the upper and lower limit for one F-ADC of the specified module.
- [P2\\_ADC\\_READ\\_LIMIT](#) returns the flags of limit-overflow and -underflow 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.
- [BURST\\_ABORT](#) aborts a running burst-measurement sequence on the specified module.
- [BURST\\_CREAD](#) copies the measurement values of a channel, stored on the specified module, into an array. The number of measurement values to be copied has to be indicated.
- [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.
- [BURST\\_CSTART](#) starts a burst-measurement sequence in the "Continuous" mode on the specified module.
- [P2\\_BURST\\_INIT](#) sets the parameters for a burst-measurement sequence on the specified module.
- [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.
- [BURST\\_READ](#) copies the measurement values of a channel into a specified array.
- [P2\\_BURST\\_READ\\_INDEX](#) returns the address in the module memory, where the last measurement values have been stored.
- [BURST\\_READ\\_PACKED](#) copies the stored measurement values of a channel into a specified array. The values are packed and the copying process is effected quickly.

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.
BURST_START	starts a burst-measurement sequence on the specified module (independent of the processor).
P2_BURST_STATUS	determines the number of burst measurements which are still to execute on the specified module.
BURST_STATUS	determines the amount of the burst-measurements which are still to be executed on the specified module.
P2_BURST_STOP	stops a running burst-measurement sequence on all specified modules at the same time.
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	P2_SEQ_WAIT waits until the sequence control has converted and stored all channels of the channel group on the specified module.
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.
READADC	reads the result of a conversion from the ADC register of the specified module.
READADCF	reads out the conversion result from an F-ADC of the specified module.
READADCF_32	reads the conversion result from 2 consecutive F-ADCs of the specified module and returns them in a single 32-bit value.
READADCF_SCONV	reads out the conversion result from an F-ADC of the specified module and starts immediately a new conversion.
READADCF_SCONV_32	reads the conversion results from the 2 F-ADCs of the specified module and returns them in a 32-bit value. Then a new conversion is started immediately.
READADC_SCONV	reads out the conversion result from an ADC of the specified module and starts immediately a new conversion.
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_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.
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.
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.
P2_READ_ADCF8	reads out the conversion results from all 8 F-ADCs of the specified module.

<code>READ_ADCF8</code>	reads out the conversion results from all 8 F-ADC of the specified module.
<code>P2_READ_ADCF8_24B</code>	reads out the conversion results from all 8 F-ADC of the specified module. The return values have a resolution of 24 bits.
<code>P2_READ_ADCF8_PACKED</code>	reads out the conversion results from all 8 F-ADC of the specified module.
<code>READ_ADCF8_PACKED</code>	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.
<code>P2_READ_ADCF_SCONV</code>	reads the conversion result from an F-ADC of the specified module and immediately starts a new conversion.
<code>P2_READ_ADCF_SCONV24</code>	reads the conversion result from an F-ADC of the specified module and immediately starts a new conversion.
<code>P2_READ_ADCF_SCONV32</code>	reads the conversion results from 2 F-ADCs of the specified module and returns them in a 32-bit value.
<code>P2_READ_ADC_SCONV</code>	reads out the conversion result from an ADC of the specified module and immediately starts a new conversion.
<code>P2_READ_ADC_SCONV24</code>	reads the conversion result from an ADC of the specified module and immediately starts a new conversion.
<code>SEQ_MODE</code>	initializes the specified module for an operation with sequential control. The operating mode and the gain factor are set (the same for all channels).
<code>SEQ_READ</code>	copies a specified amount of measurement values (16-bit each) from the module to a destination array.
<code>SEQ_READ32</code>	copies all 32 measurement values (16-bit each) from the specified module into the destination array.
<code>SEQ_READ_ONE</code>	reads out a specified measurement value (16 bit) of a measurement group on the specified module.
<code>SEQ_READ_PACKED</code>	copies an even amount of measurement values (16-bit each) in pairs from the specified module to a destination array.
<code>SEQ_READ_TWO</code>	reads out at the same time 2 consecutive measurement values (16-bit each) of a measurement group, on the specified module and returns them in a 32-bit value.
<code>SEQ_SELECT</code>	determines the channels belonging to the measurement group, which will be converted by the sequential control on the specified module.
<code>SEQ_SET_DELAY</code>	determines the settling time (waiting time between 2 measurements) of the sequential control on the specified module.
<code>SEQ_STATUS</code>	determines how many channels of the measurement group are already converted and stored by the sequential control of the specified module.
<code>P2_SET_AVERAGE_FILTER</code>	determines if the module calculates a moving average and of how many values the average is calculated.
<code>SET_GAIN</code>	sets the operating mode for a channel of the specified module, and thus the gain factor and measurement range, too.
<code>SET_MUX</code>	sets the multiplexer input of the module to a specified channel and gain.
<code>P2_SE_DIFF</code>	sets the operating mode single ended or differential for all analog inputs of the specified module.
<code>SE_DIFF</code>	sets the operating mode single ended or differential for all analog inputs on the specified module.
<code>SH_SETMODE</code>	sets the mode of the sample and hold levels.
<code>START_CONV</code>	starts the A/D conversion on the specified module.
<code>P2_START_CONV</code>	starts the conversion on the specified module.
<code>P2_START_CONVF</code>	starts the conversion on one or more F-ADCs of the specified module.
<code>START_CONVF</code>	starts the conversion of one or more F-ADCs of the specified module.
<code>SYNC_MODE</code>	determines on the specified module the type of synchronization with other modules, especially for burst-measurement sequences.
<code>P2_WAIT_EOC</code>	waits for the end of conversion on the specified module.
<code>WAIT_EOC</code>	waits, until the last A/D conversion has finished.
<code>P2_WAIT_EOCF</code>	waits until the end of conversion on all F-ADCs of the specified module.
<code>WAIT_EOCF</code>	waits until the end of conversion on all specified F-ADCs.

## Analog Outputs

P2_DAC	outputs an (analog) voltage on the channel of the specified module, that corresponds to the indicated digital value.
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.
FG_CONTROL	starts or stops the function generator (output of values) on the selected output channels of the module.
FG_DEF	For the output channel of a specified module, FG_DEF defines the start address and the size of the internal buffer for the function generator mode.
FG_DELAY	sets the output rate of function generator on the specified module.
FG_MODE	enables or disables the function generator mode on the specified module.
FG_READ_INDEX	returns the position pointer of a specified function generator.
FG_STATUS	returns the status of all function generators of the module.
FG_WRITE	transfers an even number of data of an array to a specified address in the buffer of the module.
P2_START_DAC	starts the conversion or output of all DAC on the specified module
START_DAC	starts the conversion or output of all DACs on the specified module.
WRITEDAC	writes a digital value into the output register of a DAC on the specified module. The conversion into output voltage is started by the instruction START_DAC.
P2_WRITE_DAC	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.

## Communication interface

<b>CAN</b>	
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.
EN_INTERRUPT	configures a message object of the specified module to generate an external event (interrupt) when a message arrives.
EN_RECEIVE	enables a message object on the specified module to receive messages.
EN_TRANSMIT	enables a message object on the specified module to transmit messages.
GET_CAN_REG	returns the contents of a specified register on a CAN controller on the specified module.
INIT_CAN	initializes one of the CAN controllers on the specified module and sets it into an initial status.
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.
SET_CAN_BAUDRATE	sets the baud rate on one of the controllers on the specified module and returns the status information.
SET_CAN_REG	writes a value in a register of the selected CAN controller on the specified module.
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.



### Fieldbus

CHANGED_DATA	checks, if the ndata in the output area have been changed since the user's last access to the DP-RAM
CHECK_ACCESS	returns to which areas of the DP-RAM the application has access rights.
GET_PRO_BYTE	returns a byte of a specified memory address of the DP-RAM of the fieldbus module.
GET_READ_BUFFER	copies a defined data block from the memory area of the DP-RAM into the specified destination array.
INIT_SLAVE	initializes the fieldbus slave and can only be used after power up.
REQUEST_ACCESS	requests access to the DP-RAM of the slave.
REQUEST_RELEASE_ACCESS	requests to return the access right for the DP-RAM of the slave to the fieldbus.
SET_PRO_BYTE	sets a byte in the DP-RAM of the fieldbus slave.
SET_WRITE_BUFFER	copies the data from an array into a specified memory area of the DP-RAM.

### LS-BUS

LS_DIGPROG	sets the digital channels 1...32 of the specified module of type HSM-24V on the LS bus as inputs or outputs in groups of 8 via an interface of the Pro module.
LS_DIG_IO	sets all digital outputs of the specified module HSM-24V on the LS bus to the level High oder Low and returns the status of all channels as bit pattern.
LS_DIO_INIT	initializes the specified module of type HSM-24V on the LS bus via an interface of the Pro module and returns the error status
LS_WATCHDOG_INIT	enables or disables the watchdog counter of a specified module on the LS bus via an interface of the Pro module.

### RSxxx

CHECK_SHIFT_REG	returns, if all data has been sent, which was written into the send-FIFO of the channel on the specified module.
GET_RS	reads out the controller register on the specified module.
READ_FIFO	reads a value from the input FIFO of a specified channel on the specified module.
RS485_SEND	determines the transfer direction for a specified channel on the specified module.
RS_INIT	initializes one channel on the specified module.
RS_RESET	executes a hardware reset on the specified module and deletes the settings for all channels.
SET_RS	writes a value into a specified register on the specified module.
WRITE_FIFO	writes a value into the send-FIFO of a specified channel on the specified module.

### Comparator

COMP_DIGIN_WORD	returns the current status of the threshold value comparison for all channels of the specified module.
COMP_DIGIN_WORD_DIFF	returns the current status of the threshold value comparison. The comparison is applied to the difference value of 2 channels (differential signal).
COMP_FIFO_READ	reads the last 2 x 1024 measurement values of a channel pair from the internal FIFO memory and transfers the values to 2 arrays.
COMP_FIFO_SELECT	determines the channel pair whose data is stored in the internal FIFO memory of the module.
COMP_READ	returns the current minimum or maximum measurement value of a specified channel on the module.
COMP_RESET	resets the measurement of the minimum and maximum values simultaneously for the selected channels.
COMP_SET	determines the lower and upper threshold value for a specified channel.

### Counters

CNT_CLEAR	sets the counter values of one or more counters on the specified module to the value 0 (zero).
CNT_ENABLE	enables or disables one or more counters on the specified module.
CNT_LATCH	transfers the current counter values of one or more counters on the specified module into the respective latch register(s) (= to latch).
CNT_READ16	returns the current counter value of a 16-bit counter on the specified module.
CNT_READ32	returns the current counter value of a 32-bit counter on the specified module.
CNT_READLATCH16	returns the value from the latch register of a 16-bit counter on the specified module.

CNT_READLATCH32	returns the value from the latch register of a 32-bit counter on the specified module.
CNT_SETMODE	sets the operating mode of all counters on the specified module, four edge evaluation or clock and direction input.
CO4_CLEARENABLE	enables the external input CLR of one or more counters.
CO4_GETSTATUS	returns the status of the input signals of a counter on the specified module as bit pattern.
CO4_LATCHENABLE	enables the external input LATCH of one or more counters on the specified module.
CO4_READ	returns the current counter value from the specified module.
CO4_READLATCH	returns the value of the latch register of a counter on the specified module.
CO4_RESETSTATUS	clears the status register of one or more counters on the specified module.
CO4_SETMODE	sets the count mode of a counter on the specified module.
CO4_SET_LATCHMODE	determines the mode of the latch-inputs for all counters on the specified module.
EXTLCH_ENABLE	enables or disables all latch-inputs on the specified module. The latch-inputs are selected by the corresponding counter number.
SSI_MODE	sets the modes of all SSI decoders on the specified module, either "single shot" (read out once) or "continuous" (read out continuously).
SSI_READ	returns the last saved counter value of a specified SSI counter on the specified module.
SSI_SET_BITS	sets for an SSI counter on the specified module the amount of bits which generate a complete encoder value.
SSI_SET_CLOCK	sets the clock rate (approx. 40kHz to 1 MHz) on the specified module, with which the encoder is clocked.
SSI_START	starts the reading of one or both SSI encoders on the specified module (only in mode "single shot").
SSI_STATUS	returns the current read-status on the specified module for a specified decoder.

## Data Storage

MEDIA_RD_BLK_F	copies an amount of data blocks with FLOAT values from one file of the storage medium in the specified module to an array.
MEDIA_RD_BLK_L	copies an amount of data blocks with LONG values from one file of the storage medium in the specified module to an array.
MEDIA_RD_FILEINFO	initializes the glue-logic on the specified module and returns the file information (start and end sector) into an array.
MEDIA_WR_BLK_F	copies a number of FLOAT data blocks from an array into one file on the storage medium in the specified module.
MEDIA_WR_BLK_L	copies a number of LONG data blocks from an array into one file on the storage medium in the specified module.
RTC_GET	returns date and time from the real-time clock of the specified module. Invalid values are not accepted.
RTC_SET	sets date and time on the real-time clock of the specified module. Invalid values are not accepted.

## Digital Inputs/Outputs

DIGIN_LONG_F	returns the status of the inputs (bits 31...00) of the specified module as bit pattern.
DIGIN_WORD1	returns the status of the inputs 0...15 of the specified module as bit pattern.
DIGIN_WORD2	returns the status of the inputs 16...31 of the specified module as bit pattern.
DIGOUT	sets a single output of the specified module to the level "high" or "low". All other outputs remain unchanged.
DIGOUT_BITS_F	sets the specified outputs of the specified module to the levels "high" or "low".
DIGOUT_F	sets a single output of the specified module to the level "high" or "low". All other outputs remain unchanged.
DIGOUT_LONG_F	With the 32-bit value the instruction DIGOUT_LONG_F sets or clears all outputs on the specified module.
DIGOUT_WORD1	sets the digital outputs 0...15 on the specified module simultaneously to the specified levels.
DIGOUT_WORD2	sets the digital outputs 16...31 on the specified module simultaneously to the specified levels.
DIGPROG1	programs the digital channels 0...15 of the specified module as input or output.
DIGPROG2	programs all channels 16...31 of the specified module as inputs or outputs.

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.
DIG_READLATCH1	returns the lower 16 bits (bit 0...bit 15) from the latch register for the digital inputs of the specified module.
DIG_READLATCH2	returns the upper 16 bits (bit 16... bit 31) from the latch register for the digital inputs of the specified module.
DIG_WRITELATCH1	writes a value into the lower 16 bits (bit 0...15) of the latch register for the digital outputs of the specified module.
DIG_WRITELATCH2	writes a value into the upper 16 bits (bit 16...31) of the latch register for the digital outputs of the specified module.
DIG_WRITELATCH32	writes a 32-bit value into the long-word (bits 31...0) of the latch on the specified module.
GET_DIGOUT_LONG	returns the contents of the output-latch (register for digital outputs) on the specified module.
GET_DIGOUT_WORD1	returns the lower word (bits 0...15) of the output-latch (register for digital outputs) on the specified module.
GET_DIGOUT_WORD2	returns the upper word (bits 16...31) of the output-latch (register for digital outputs) of the specified module.
PWM_ENABLE	enables or disables all internal counters of the specified module. The counters are selected by the corresponding PWM output number.
PWM_OUT	sets a specified PWM output channel on the specified module to the level "high" or "low".
PWM_SET	makes the settings for a specified PWM output channel on the specified module.

### System

CHECKLED	returns the status of the green LED (on top of the front panel) of the module.
P2_CHECK_LED	returns the status of the LED (on top of the front panel) of the module.
CPU_DIGIN	Processor T9 and T10 only. CPU_DIGIN returns, whether a falling edge arose at the input Digin 0 of the processor module since the last call of the instruction.
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.
EVENTENABLE	enables or disables an external event input on the module. With a signal at this input a cycle of an ADbasic process can be controlled.
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.
RESETWATCHDOGTIMER	sets the watchdog counter of the CPU module to the start value. The counter remains enabled.
SETLED	switches the green LED (on top of the front panel) on or off.
P2_SET_LED	switches the LED (on top of the front panel) on or off.
STARTWATCHDOG	activates the watchdog counter of the CPU module and sets its start value.
STOPWATCHDOG	disables the watchdog counter of the CPU module.
SYNCALL	starts a specified action synchronically on all modules which have been activated before with SYNCENABLE.
SYNCENABLE	enables or disables the synchronizing option on the specified module.
SYNCSTAT	returns the settings of the synchronizing option of the specified module.
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 on the specified module.
P2_SYNC_STAT	returns the settings of the synchronizing option of the specified module.

### Temperature-Inputs

PT100_DIG_TO_R	calculates the resistance in Ohm from the measured digital value of a PT100 sensor.
----------------	---

PT100_DIG_TO_TEMP	calculates the temperature in degrees Celsius or Fahrenheit from the measured digital value of a PT100 sensor.
TCJ_DIG_TO_TEMP	TCK_DIG_TO_TEMP calculates the temperature in degrees Celsius or Fahrenheit from the measured digital value of a thermo couple type K.
TCK_DIG_TO_TEMP	calculates the temperature in degrees Celsius or Fahrenheit from the measured digital value of a thermo couple type K.
TC_READ_B	TC_READ_E returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$ / $^{\circ}\text{F}$ ) of a specified channel on the module.
TC_READ_E	returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$ / $^{\circ}\text{F}$ ) of a specified channel on the module.
TC_READ_J	returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$ / $^{\circ}\text{F}$ ) of a specified channel on the module.
TC_READ_K	returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$ / $^{\circ}\text{F}$ ) of a specified channel on the module.
TC_READ_N	returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$ / $^{\circ}\text{F}$ ) of a specified channel on the module.
TC_READ_R	returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$ / $^{\circ}\text{F}$ ) of a specified channel on the module.
TC_READ_S	returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$ / $^{\circ}\text{F}$ ) of a specified channel on the module.
TC_READ_T	returns the thermoelectric voltage ( $\mu\text{V}$ ) or the temperature ( $^{\circ}\text{C}$ / $^{\circ}\text{F}$ ) of a specified channel on the module.
TC_SELECT	sets the thermocouple channels via multiplexer to the analog output of the module.
TC_SET_RATE	sets the sample rate for the specified module.