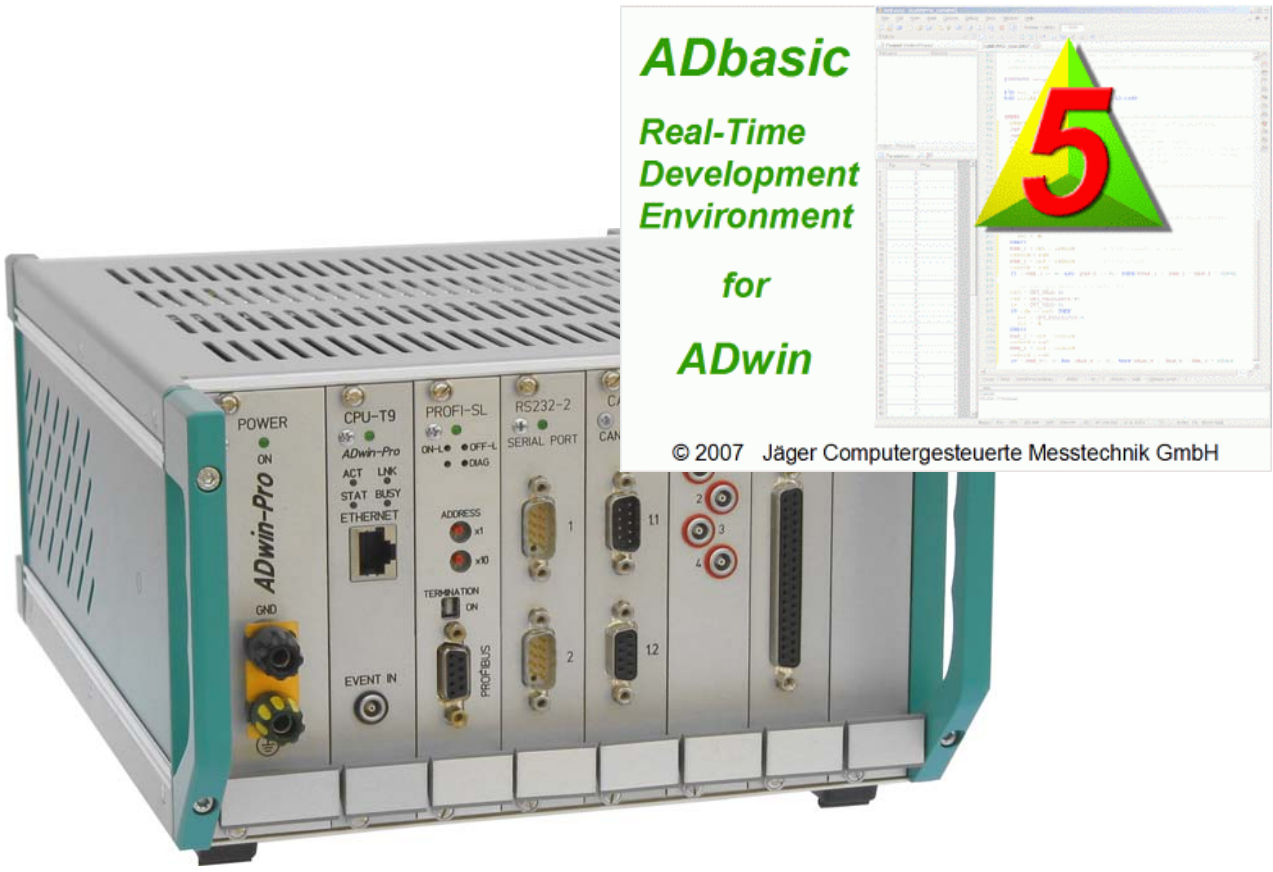
JÄGER
Computergesteuerte
Messtechnik GmbH

ADwin-Pro I

System specifications
Programming in *ADbasic*



ADwin-Pro software, Manual Feb. 2017

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

Hotline:	+49 6251 96320
Fax:	+49 6251 5 68 19
E-Mail:	info@ADwin.de
Internet	www.ADwin.de

JÄGER
Computergesteuerte
Messtechnik GmbH

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

Table of contents

Typographical Conventions

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

You find a "note" next to

– information, which absolutely have to be considered in order to guarantee an error free operation.

– advice for efficient operation.

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

<C:\ADwin\ …>

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

Program text

Program commands and user inputs are characterized by the font `Courier New`.

Var_1

Source code elements such as commands, variables, comments and other text are characterized by the font `Courier New` and are printed in color.

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.

Qualified personnel

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 acivities, while recognizing and avoiding any possible dangers.*
> *(Definition of qualified personnel as per VDE 105 and ICE 364).*

Availability of the documents

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.

Legal information

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.

The Program ADpro.exe

1 The Program ADpro.exe

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

– Show the modules on an *ADwin-Pro II* system a well as information on the modules.

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

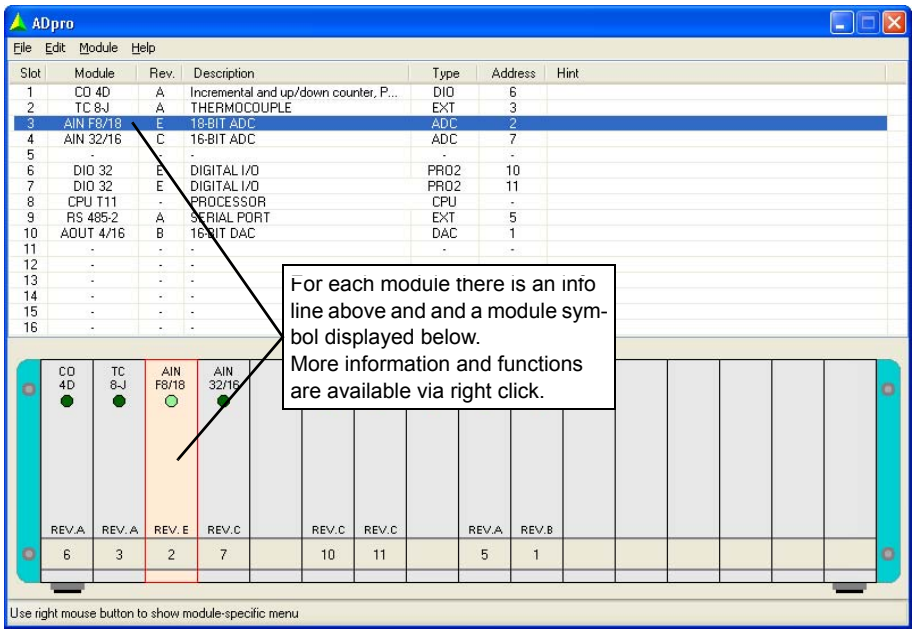
  With Pro I modules, the module address is set manually; here the address can be shown only.

– Check the function of Pro I and Pro II modules: analog input / output modules, digital and counter modules, some bus modules.

– Calibrate Pro I and Pro II modules (analog input / output modules).

  The calibration meets lower demands only.

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



Notes

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

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

The recognition of module types, one of the tasks of ADpro.exe, is sometimes requested as function in *ADbasic*, too. But it appeared that the user's organisational effort exceeds the benefits by far: the required module information would have to be regularly updated, evaluated, and added manually to the *ADbasic* source code. Therefore, the function „recognition of module types" is only available in ADpro.exe, but not in *ADbasic*.

2 *ADbasic* instruction for *ADwin-Pro I* modules

This section contains all instructions to access *ADwin-Pro I* modules. 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.

2.1 Pro I: All Modules

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

Some instructions for Pro I modules need the parameter `mod_class` that determines the module class. Use the following constants for this parameter:

`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

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

Notes

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

See also

SetLED

Valid for

(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-4/16 Rev. B, AIn-F-8/12 Rev. A, AIn-F-8/14 Rev. B, AIn-F-8/16 Rev. A, AIn-F-8/16 Rev. B, 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-D, CO4-I, CO4-T, COMP-16 Rev. A, CPU-T10, CPU-T9, DIO-32 Rev. A, DIO-32 Rev. B, Inter-SL, LS-2 Rev. A, OPT-16 Rev. A, OPT-16 Rev. B, Profi-DP-SL, Profi-IRT-CU, Profi-IRT-FO, 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
```

CPU_Digin

Processor T9 and T10 only. **CPU_Digin** returns, wether 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

ret_val	LONG	Flag, if a falling edge has been detected at the input Digin 0: 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)

Valid for

CPU-T10, CPU-T9

Example

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

Init:
  'Read and thus reset the flag
  dummy = CPU_Digin()

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

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

module	Specified module address (1…255).	LONG
mod_class	Module class: One of the constants ad, da, dio or ext.	LONG
value	0: disable external event signal (default). 1: enable external event signal.	LONG

Notes

One high-priority *ADbasic* process (more precisely: its cyclic section **Event:**), may be called by an external event signal, e.g. to synchronize it with an external process (see *ADbasic* manual). The *ADbasic* process is to be set as external controlled in *ADbasic*. (see *ADbasic* manual, Process Options).

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 external controlled process responds (if existing).

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

- / -

Valid for

CNT-16/16(-I), CNT-16/32(-I), CNT-8/32(-I), CNT-PW4(-I), CNT-VR4(-I), CNT-VR4L(-I), CO4-D, CO4-I, CO4-T, DIO-32 Rev. A, 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:
  Rem …
```

ResetWatchdogTimer

ResetWatchdogTimer resets 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	1600 ms

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.

See also

StartWatchdog, StopWatchdog

Valid for

CPU-T10, CPU-T9

Example

```
#Include ADwinPro_All.Inc

Init:
  StartWatchdog()           'Activate watchdog

Event:
  ResetWatchdogTimer()      'Reset watchdog continuously
  Rem …

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 ret_val	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

Valid for

(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-4/16 Rev. B, AIn-F-8/12 Rev. A, AIn-F-8/14 Rev. B, AIn-F-8/16 Rev. A, AIn-F-8/16 Rev. B, 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-D, CO4-I, CO4-T, COMP-16 Rev. A, CPU-T10, CPU-T9, DIO-32 Rev. A, DIO-32 Rev. B, Inter-SL, LS-2 Rev. A, OPT-16 Rev. A, OPT-16 Rev. B, Profi-DP-SL, Profi-IRT-CU, Profi-IRT-FO, 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:
  Rem …

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
```

StartWatchdog

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 value 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 **ResetWatchdogTimer**).

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

See also

ResetWatchdogTimer, StopWatchdog

Valid for

CPU-T10, CPU-T9

Example

```
#Include ADwinPro_All.Inc
Init:
  StartWatchdog()           'Activate watchdog
```

StopWatchdog

`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

ResetWatchdogTimer, StartWatchdog

Valid for

CPU-T10, CPU-T9

Example

```
#Include ADwinPro_All.Inc
Init:
  StartWatchdog()           'Activate watchdog
  Rem …

Event:
  ResetWatchdogTimer()      'Reset watchdog
  Rem …

Finish:
  StopWatchdog()            'Disable watchdog
```

SyncAll

SyncAll starts a specified action synchronically on all modules, which have bbeen 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	Ersetzter Befehl
Analog input	Start A/D conversion on all ADCs: **SyncEnable**(…,…,1) or Start burst measurement sequence: **SyncEnable**(…,…,3) and only for modules Pro-AIn-F-x/14	**Start_Conv** / **Start_ConvF** **Burst_Start**
Analog output	Start D/A conversion with the value from the DAC register, see **WriteDAC**.	**Start_DAC**
Digital input	Transfer current status of the inputs into the input latch register. Read values with **Dig_ReadLatch1/2**.	**Dig_Latch**
Digital output	Read values from the output latch register and transfer to the digital output. See **Dig_WriteLatch1/2**.	**Dig_Latch**
Counter	Transfer current counter values into the counter latch register. Read values with **Cnt_ReadLatch16**, **Cnt_ReadLatch32**, **CO4_ReadLatch**.	**Cnt_Latch**

See also

SyncEnable, SyncStat

Valid for

(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-4/16 Rev. B, AIn-F-8/12 Rev. A, AIn-F-8/14 Rev. B, AIn-F-8/16 Rev. A, AIn-F-8/16 Rev. B, 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-D, CO4-I, CO4-T, DIO-32 Rev. A, 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
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,enable)
```

Parameters

module	Specified module address (1…255).	LONG
mod_class	Module class: One of the constants ad, da, dio, cpu or ext.	LONG
enable	Setting of the synchronizing option: 0 : disabled. 1: enabled (for "normal" action; see **SyncAll**). 3: enabled for burst measurement sequence (only for the modules Pro-AIn-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

Valid for

(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-4/16 Rev. B, AIn-F-8/12 Rev. A, AIn-F-8/14 Rev. B, AIn-F-8/16 Rev. A, AIn-F-8/16 Rev. B, 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-D, CO4-I, CO4-T, DIO-32 Rev. A, 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
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 **SyncAll**). 3 : enabled for burst-measurement sequences (only for the modules Pro-AIn-F-x/14).	LONG

See also

SyncAll, SyncEnable

Valid for

(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-4/16 Rev. B, AIn-F-8/12 Rev. A, AIn-F-8/14 Rev. B, AIn-F-8/16 Rev. A, AIn-F-8/16 Rev. B, 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-D, CO4-I, CO4-T, DIO-32 Rev. A, 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
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
```

2.2 Pro I: Analog Input Modules

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

Analog Inputs with Multiplexer

Analog Inputs with Fast ADC and Burst Sequence

Analog Inputs with Fast ADC

- ADCF (page 22)
- ReadADCF (page 40)
- Read_ADCF4 (page 41)
- Read_ADCF8 (page 42)
- Read_ADCF4_Packed (page 43)
- Read_ADCF8_Packed (page 44)
- ReadADCF_32 (page 45)
- ReadADCF_SConv (page 46)
- ReadADCF_SConv_32 (page 47)
- Start_ConvF (page 68)
- Sync_Mode (page 70)
- Wait_EOCF (page 73)

In the Instruction List sorted by Module Types (annex A.2), you will find, which of the functions corresponds to the *ADwin-Pro I* 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); with 12-bit ADCs, the 4 LSBs are always 0, with 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) with AIn-x/16 Rev. B or higher: 14µs		Start A/D conversion		Wait for end of conversion		Read out the converted value

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

- Very short cycle times: **Processdelay** < 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.
- You want to set the gain.

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

Valid for

(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. B, AIn-8/16 Rev. C, AOut-16/8-12

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:

Set_Mux	→	…	→	Start_Conv	→	Wait_EOC	→	
Set the multiplexer to the specified input		Wait for settling of the multiplexer 5 µs		Start A/D conversion		Wait for end of conversion		
			→	Start_Conv	→	Wait_EOC	→	ReadADC
				Start A/D conversion		Wait for end of conversion		Read out the converted value

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

- Very short cycle times: **Processdelay** < 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.
- You want to set the gain.

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

Valid for

(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:

Start_ConvF	→	Wait_EOCF	→	ReadADCF
Start A/D conversion		Wait for end of conversion		Read out the converted value

See also

Start_ConvF, Wait_EOCF, ReadADCF, Read_ADCF4, Read_ADCF8, Read_ADCF4_Packed, Read_ADCF8_Packed

Valid for

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

Example

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

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

Burst_Abort

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…1,048,575 = 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

Valid for

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

Example

```
REM Measurement with high priority process, reading out in the
REM low-priority section Finish:.
REM Measurement will abort with a trigger signal at DIO32 module 1
#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 num_data 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
    num_data=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?
      num_data=Burst_Abort(ainadr)'Abort measurement sequence /
                            'number of measurement values
      End                   'End of progam
    EndIf
    If (num_data=0) Then End 'Are all measurements executed?:
                            'terminate
  EndIf

Finish: 'Get measurement values in low-priority section
  Burst_Read(ainadr,1,1,num_data,Data_1,1)
```

Burst_CRead

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,count,array[],arr_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
count	Number of the measurement values to be transferred (1…n), divided by 2.	LONG
array[]	Destination array, into which the measurement values are transferred; no FIFO array allowed.	LONG
arr_idx	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` + `count` 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

Valid for

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

Example

```
#Include ADwinPro_All.Inc
#Define count 10000
#Define module 1
#Define sampleperiod 20    '2000 kHz measurement rate
                           '[=1/(25ns*20)]
Dim Data_1[count] As Long

Init:
  'Address, mode, meas. rate, number of measurements
  Burst_Init(module,1,sampleperiod,count)
  'Start continuous burst-measurement
  Burst_CStart(module)
  Processdelay=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,count,Data_1,1)
```

Burst_CStart

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 occured.

See also

Burst_Init, Burst_CRead, Burst_Status, Set_Gain, Sync_Mode

Valid for

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

Example

```
#Include ADwinPro_All.Inc
#Define count 10000
#Define module 1
#Define sampleperiod 20    '2000 kHz measurement rate
                           '[=1/(25ns*20)]
Dim Data_1[count] As Long

Init:
  Burst_Init(module,1,sampleperiod,count) 'Address, mode,
                          'meas. rate, number of measurements
  Burst_CStart(module)     'Start continuous burst-measurement
  Processdelay=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,count,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

module	Specified module address (1…255).	LONG
mode	Measurement mode (1…3; with a Pro-AIn-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

mode .	Channel no.	max. amount of measurement values per channel:
n.	1…2^(n-1)	2^(21-n) - 1
1.	1	2^{20} - 1 = 1,048,575 = 0FFFFFH
2.	1, 2	2^{19} - 1 = 524,287 = 7FFFFH
3.	1…4	2^{18} - 1 = 262,143 = 3FFFFH
4.	1…8	2^{17} - 1 = 131,071 = 1FFFFH

pulses	determines the period duration of a measurement sequence as number of time intervals: period duration = pulses * 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
samples	The amount of measurements per channel to be executed (the maximum value for samples is determined by mode).	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 the same in the various burst-measurement sequences.

The instructions **ADCF** and **Start_ConvF** 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

Valid for

AIn-F-4/14 Rev. B, AIn-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
#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)'Determing 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

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 (1…n).	LONG
array[]	Destination array, into which the measurement values are transferred; no FIFO array allowed.	LONG
array_idx	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 + count-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 using 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

Valid for

AIn-F-4/14 Rev. B, AIn-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
#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.
                            'count, 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.	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 + count in order to get all measurement values.

Address in the destination array[]	Higher word (bits 31…16)	Lower word (bits 15…0)
array_idx	Meas. value 2	Meas. value 1
array_idx + 1	Meas. value 4	Meas. value 3
…	…	…
array_idx + count	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 using 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

Valid for

AIn-F-4/14 Rev. B, AIn-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
#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

Valid for

AIn-F-4/14 Rev. B, AIn-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
#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

Valid for

AIn-F-4/14 Rev. B, AIn-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
#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

Valid for

(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, AOut-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-8/16Rev.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 T9 or T10; for T11 see chapter 5.2.4).

```
  Start_Conv(1)             'Start AD conversion
  Par_80 = Read_Timer()
  Do
  Until (Read_Timer() - Par_80 > 560) 'Wait 25ns*560=14µs
```

ReadADC_SConv

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-AIn-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

Valid for

(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. B, AIn-8/14 Rev. A, AIn-8/16 Rev. A, AIn-8/16 Rev. B, AIn-8/16 Rev. C

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
  Rem Wait 3µs (12-bit ADC) or
  Rem 14µs (AIn-8/16 Rev. B, AIn-32/16 Rev. B) for the settling
  Rem 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_ConvF, Wait_EOCF, ReadADCF_32, ReadADCF_SConv, ReadADCF_SConv_32, Read_ADCF4, Read_ADCF8, Read_ADCF4_Packed, Read_ADCF8_Packed

Valid for

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

Example

```
#Include ADwinPro_All.Inc
Dim val1 As Long            'Declaration

Event:
  Start_ConvF(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

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

module	Specified module address (1…255).	LONG
array[]	Destination array, where conversion results are saved.	LONG
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 subsequentially 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_ConvF, ReadADCF, Read_ADCF8, Read_ADCF4_Packed, Read_ADCF8_Packed, ReadADCF_SConv, Wait_EOCF

Valid for

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

Example

```
#Include ADwinPro_All.Inc
Dim value[4] As Long        'Array for conversion results

Init:
  Start_ConvF(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_ConvF(1,0Fh)        'Start new AD conversion
```

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

module	Specified module address (1…255).	LONG
array[]	Destination array, where conversion results are saved.	LONG
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 subsequentially 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_ConvF, Wait_EOCF, Read_ADCF4, Read_ADCF4_Packed, Read_ADCF8_Packed, ReadADCF_SConv

Valid for

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

Example

```
#Include ADwinPro_All.Inc
Dim value[8] As Long        'Array for conversion results

Init:
  Start_ConvF(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_ConvF(1,0FFh)       'Start new AD conversion
```

Read_ADCF4_Packed

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

module	Specified module address (1…255).	LONG
array[]	Destination array, where conversion results are saved.	LONG
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

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

Valid for

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

Example

```
#Include ADwinPro_All.Inc
Dim value[2] As Long       'Array for conversion results

Init:
  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_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

module	Specified module address (1…255).	LONG
array[]	Destination array, where conversion results are saved.	LONG
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 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
index+2	F-ADC 6	F-ADC 5
index+3	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_ConvF, Wait_EOCF, Read_ADCF4, Read_ADCF8, Read_ADCF4_Packed, ReadADCF_SConv

Valid for

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

Example

```
#Include ADwinPro_All.Inc
Dim value[4] As Long       'Array for conversion results

Init:
  Start_ConvF(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_ConvF(1,0FFh)      'Start new AD conversion
```

ReadADCF_32

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

Valid for

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

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_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

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_32, Start_ConvF, Wait_EOCF

Valid for

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

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

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

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

Valid for

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

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
```

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

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

Notes

In operating mode single-ended, 32 inputs are available, in 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

Valid for

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

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

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	Gain	Measure-ment range
mode	2^n	±10V / 2^n
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_ConvF, Wait_EOCF

Valid for

AIn-F-4/14 Rev. B, AIn-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 analog input 4
```

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 AIn-8/16 Rev. B, AIn-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-AIn-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

Valid for

(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, AOut-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-32/16Rev.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

After power-up mode 0 is active.

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 the channel group 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 **ADC**).
1 "single shot":	The sequential control is started by **Start_Conv**; it ends as soon as each of the selected channels is converted once. The end of the sequential control is queried with **Seq_Status** and measurement values are read e.g. with **Seq_Read**.
3 "continuous":	The sequential control continuously converts new measurement values at the selected channels. The conversion is started with **Start_Conv**. **Seq_Read** reads the most recent measurement values.

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 (>3kΩ), the predefined settling time of the multiplexer will not be sufficient for an exact measurement. You can change the waiting time until the next conversion with the instruction **Seq_Set_Delay**.

See also

Seq_Select, Seq_Set_Delay, Seq_Status, Seq_Read

Valid for

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

Finish:
  Seq_Mode(module,0,0)       'reset to standard mode
```

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,count,array[],array_idx)
```

Parameters

module	Specified module address (1…255).	LONG
count	Number of measurement values being read (1…32). The number of values should not exceed the number of channels in the channel group.	LONG
array[]	Destination array where the measurement values are transferred.	LONG
array_idx	Destination start index: 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** and a channel group was defined with **Seq_Select**.

If more measurement values are read than there are channels in the channel group, the surplus values are undefined and should be discarded.

The measurement values of the channel group 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

Valid for

AIn-16/14-C Rev. A, AIn-32/14 Rev. A, AIn-32/16 Rev. C, AIn-8/14 Rev. A, AIn-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, gain factor 1
  Seq_Select(1,55555555h)   'Measure all odd-numbered channels
                            'of an AIN-32/.. module
  Start_Conv(1)             'Start measurement sequences in
                            '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

Finish:
  Seq_Mode(1,0,0)           'reset to standard mode
```

Seq_Read_One

Seq_Read_One reads out a specified measurement value (16 bit) of a channel 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 channel group (1 … 32).	LONG
ret_val	The last saved measurement value of the channel with the position idxno in the channel group.	LONG

Notes

You can only reasonably use this instruction if the sequential control of the module has been activated before by **Seq_Mode** and a channel group was defined with **Seq_Select**.

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

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

See also

Seq_Mode, Seq_Select, Seq_Status, Seq_Read

Valid for

AIn-16/14-C Rev. A, AIn-32/14 Rev. A, AIn-32/16 Rev. C, AIn-8/14 Rev. A, AIn-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(1,0FFFFFFFFh)    'Measure all inputs 1…32 of an
                              'AIN-32/.. module

Event:
  Start_Conv(1)               'Start measurement sequence in
                              'single-shot mode
  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

Finish:
  Seq_Mode(1,0,0)             'reset to standard mode
```

Seq_Read_Two

Seq_Read_Two reads out at the same time 2 consecutive measurement values (16-bit each) of a channel 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 channel 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** and a channel group was defined with **Seq_Select**.

The index number must not be interpreted as the number of a channel, but it indicates the position of two channels in the channel 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 channel group with the channels (1,3,7,11,12,15) means that the channels 7 and 11 are read. 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(1,0FFFFFFFFh)  'Measure all 32 inputs of the
                            'AIN-32/..

Event:
  Start_Conv(1)             'Start measurement sequence in
                            'single-shot mode
  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)

Finish:
  Seq_Mode(1,0,0)           'reset to standard mode
```

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

module	Specified module address (1…255).	LONG
length	Amount of the measurement value pairs to read (1…16). The number of values should not exceed the number of channels in the channel group.	LONG
array[]	Destination array where the measurement value pairs are transferred.	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** and a channel group was defined with **Seq_Select**.

If more measurement values are read than there are channels in the channel group, the surplus values are undefined and should be discarded. If a channel group consists of an odd number of channels inevitably one surplus value has to be read.

The measurement values of the channel 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

Valid for

AIn-16/14-C Rev. A, AIn-32/14 Rev. A, AIn-32/16 Rev. C, AIn-8/14 Rev. A, AIn-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(1,0AAAAAAAAh)  'Measure all even-numbered channels
                            'of a AIN-32/.. module
  Start_Conv(1)             'Start measurement sequences in
                            '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

Finish:
  Seq_Mode(1,0,0)           'reset to standard mode
```

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

module	Specified module address (1…255).	LONG
array[]	Destination array where the measurement values are transferred.	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** and a channel group was defined with **Seq_Select**.

If more measurement values are read than there are channels in the channel group, the surplus values are undefined and should be discarded.

The measurement values of the channel 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

Valid for

AIn-16/14-C Rev. A, AIn-32/14 Rev. A, AIn-32/16 Rev. C, AIn-8/14 Rev. A, AIn-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(1,0FFFFFFFFh)  'Measure all channels with an
                            'AIN-32/..
  Start_Conv(1)             'Start measurement sequences in
                            '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

Finish:
  Seq_Mode(1,0,0)           'reset to standard mode
```

Seq_Select

`Seq_Select` determines the channels belonging to the channel group, which will be converted by the sequential control on the specified module.

Syntax

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

Parameters

`module`	Specified module address (1…255).	LONG
`pattern`	Bit pattern to select the channels for conversion as channel group.	LONG

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

Notes

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 lowest number first. Not selected channels are skipped, and there are no measurement values assigned to them.

In the channel group, you can combine any of the 32 channels of the module. The channels of a channel 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. Thus, if a channel group consists of 3 channels, you can read 3 measurement values only.

On 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

Valid for

AIn-16/14-C Rev. A, AIn-32/14 Rev. A, AIn-32/16 Rev. C, AIn-8/14 Rev. A, AIn-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 Single Shot
  Rem 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

Finish:
  Seq_Mode(1,0,0)           'reset to standard mode
```

Seq_Set_Delay

Seq_Set_Delay determines the settling time of the sequential control on the specified module.

Syntax

```
#Include ADwinPro_All.Inc
Seq_Set_Delay(module,time)
```

Parameters

module	Specified module address (1…255).	LONG
time	Number of time units of the variable part of the settling time of the sequential control: 0: Standard settling time (default). 1…2047: Settling time in time units of 25ns.	LONG

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

Notes

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

The waiting time is calculated according to the following formula:

$$\text{waiting time} = \text{conversion time} + 25\ \text{ns} \cdot \text{time}$$

The default value of time will make the waiting time equal to the multiplexer settling time of the module. Example: If the module's settling time is 3µs and conversion time is 0.5µs, the setting of 0 for time will equal the value 100.

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

See also

Seq_Mode, Seq_Read

Valid for

AIn-16/14-C Rev. A, AIn-32/14 Rev. A, AIn-32/16 Rev. C, AIn-8/14 Rev. A, AIn-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
                            'single-shot mode
  Rem 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

Finish:
  Seq_Mode(1,0,0)           'reset to standard mode
```

Seq_Status

Seq_Status determines how many channels of the channel 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

module	Specified module address (1…255).	LONG
ret_val	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 channel group, defined by **Seq_Select**.

For instance, return value 4 in a channel 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

Valid for

AIn-16/14-C Rev. A, AIn-32/14 Rev. A, AIn-32/16 Rev. C, AIn-8/14 Rev. A, AIn-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(1,0FFFFFFFFh)  'Measure all 32 inputs of the
                            'AIN-32/..

Event:
  Start_Conv(1)             'Start measurement sequence in
                            'single-shot mode
  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

SH_SetMode sets the mode of the sample and hold levels.

Syntax

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

Parameters

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

Notes

In "sample" mode, 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 µV/ms at 25°C operating temperature, the acquisition time to 0.01 % is approx. 20 µs.

See also

- / -

Valid for

(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
```

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

Valid for

(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, AOut-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 (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
  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 (with T9 or T10; for T11 see chapter 5.2.4):

```
  Dim time As Long
  time = Read_Timer()       'Determine current timer rate
  Do
  Until (Read_Timer()-time>120) 'Wait 25ns*120=3µs
```

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

```
  Until (Read_Timer()-time>560)'Wait 25ns*560=14µs
```

Start_ConvF

`Start_ConvF` starts the conversion of one or more F-ADCs of the specified module.

Syntax

```
#Include ADwinPro_All.Inc
Start_ConvF(module,adc_no)
```

Parameters

module	Specified module address (1…255).	LONG
adc_no	Bit pattern that determines the ADCs whose conversion is to be started (see table): 1: start conversion. 0: do not start conversion.	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

Valid for

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

Example

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

Event:
  Start_ConvF(1,1)         'Start AD conversion
  Wait_EOCF(1,1)           'Wait for end of conversion
  value = ReadADCF(1,1)    'Read the value from the ADC
```

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

module	Specified module address (1…255).	LONG
mode	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_ConvF**). 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_ConvF**) 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 measrurement 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 10,000 measurements, then 10,000 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

Valid for

AIn-F-4/14 Rev. B, AIn-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
  Processdelay=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

Valid for

(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, AOut-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 multiplexer[1]
  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 (with T9 or T10; for T11 see chapter 5.2.4):

```
  Dim time As Long
  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

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_ConvF, ReadADCF, Read_ADCF4, Read_ADCF8, Read_ADCF4_Packed, Read_ADCF8_Packed

Valid for

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

Example

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

Event:
  Start_ConvF(1,1)         'Start AD conversion
  Wait_EOCF(1,1)           'Wait for end of conversion
  value = ReadADCF(1,1)    'Read value from ADC
```

2.3 Pro I: Output Modules

This section describes instructions, which apply to Pro I analog output modules:

- DAC (page 75)
- FG_Control (page 76)
- FG_Def (page 78)
- FG_Delay (page 79)
- FG_Mode (page 80)
- FG_Read_Index (page 82)
- FG_Status (page 83)
- FG_Write (page 84)
- Start_DAC (page 86)
- WriteDAC (page 87)

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

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

DAC

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 (1…8) of the output.	LONG
value	Value to output (0…65535).	LONG

This procedure is characterized by a sequence of several commands:

WriteDAC	→	Start_DAC
Transfer digital value into DAC register		Start D/A conversion

See also

Start_DAC, WriteDAC

Valid for

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
#Define set_to Par_1       'set point
#Define gain FPar_2        'Gain
#Define dev Par_3          'control deviation
#Define actuate Par_4      'actuating value

Event:
  dev = set_to - ADC(1,1)   'Calculate control deviation
  actuate = dev * gain      'Calculate actuating value
  DAC(1,1,actuate)          'Output of actuating value
```

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 `run_mode`.	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 fom 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

Valid for

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)   'dac_no=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, count=100,
                           'array=values, array_idx=1
  FG_Delay(1, 1, 80000)    'dac_no=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,dac_no,startadr,memsize)
```

Parameters

module	Specified module address (1…255).	LONG
dac_no	Number (1…4) of the output channel.	LONG
startadr	Start address of the buffer (1…0FFFFFh).	LONG
memsize	Size of the buffer (1…0FFFFFh) 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 1,048,575 (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

Valid for

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)   'dac_no=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, count=100,
                           'array=values, array_idx=1
  FG_Delay(1, 1, 80000)    'dac_no=1, scale=80000 (= 1kHz output
                           'rate); with 100 values there is a
                           'sawtooth period of 100ms
```

FG_Delay

FG_Delay sets the output rate of function generator on the specified module.

Syntax

```
#Include ADwinPro_All.Inc
FG_Delay(module,dac_no,scale)
```

Parameters

module	Specified module address (1…255).	LONG
dac_no	Number (1…4) of the output channel.	LONG
scale	Scale factor (80…2,68·10^8=2^{29}-1) for setting the output rate using the formula: Output rate = 80 MHz / 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

Valid for

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)   'dac_no=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, count=100,
                           'array=values, array_idx=1
  FG_Delay(1, 1, 80000)    'dac_no=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

module	Specified module address (1…255).	LONG
mode	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, FG_Write**	**LowInit:**
– Start function generator:	**FG_Control**	**Init:**
– Query if needed:	**FG_Read_Index, FG_Status**	**Init:** **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

Valid for

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)   'dac_no=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, count=100,
                           'array=values, array_idx=1
  FG_Delay(1, 1, 80000)    'dac_no=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,dac_no)
```

Parameters

module	Specified module address (1…255).	LONG
dac_no	Number (1…4) of the output channel.	LONG
ret_val	Memory address (1…0FFFFFh) 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

Valid for

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)   'dac_no=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, count=100,
                           'array=values, array_idx=1
  FG_Delay(1, 1, 80000)    'dac_no=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

FG_Status returns the status of all function generators of the module.

Syntax

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

Parameters

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

Valid for

AOut-4/16 Rev. C

Example

```
#Include ADwinPro_All.Inc

LowInit:
  FG_Mode(1, 1)              'enable function generator mode
  FG_Def(1, 1, 200, 100)     'dac_no=1, startadr=200, memsize=100
  Rem …

Init:
  FG_Control(1,01b, 0)       'immediately start output at DAC 1

Event:
  Rem …

Finish:
  Rem …
  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_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_Write(module,startadr,count,array[],array_idx)
```

Parameters

module	Specified module address (1…255).	LONG
startadr	Start address (0…0FFFFFh) in the buffer. Beginning here, data are stored.	LONG
count	Amount (2…0FFFFEh) of array elements to be transferred. The amount must be even numbered.	LONG
array[]	Source array whose values are transferred to the memory.	LONG
array_idx	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 count must correspond to the settings made under **FG_Def**.

The source array must have count+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

Valid for

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)   'dac_no=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, count=100,
                           'array=values, array_idx=1
  FG_Delay(1, 1, 80000)    'dac_no=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:
  Rem …
```

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

Valid for

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

WriteDAC writes a digital value into the output register of a DAC on the specified module. The conversion into output voltage is started by **Start_DAC**.

Syntax

```
#Include ADwinPro_All.Inc
WriteDAC(module,dac_no,value)
```

Parameters

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

Notes

Start_DAC starts the conversion of the register value into an output voltage.

See also

Start_DAC, DAC

Valid for

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
```

WriteDAC32

WriteDAC32 writes 2 digital values into 2 output registers of a DAC on the specified module. The conversion into output voltage is started by **Start_DAC**.

Syntax

```
#Include ADwinPro_All.Inc
WriteDAC32(module,dac_no,value)
```

Parameters

module	Specified module address (1…255).	LONG
dac_no	Number (0…3) to select an output pair: 0: outputs 1 and 2 1: outputs 3 and 4 2: outputs 5 and 6 3: outputs 7 and 8	LONG
value	Value to output (0…4095 with 12-bit DAC, 0…65535 with 16-bit DAC).	LONG

Notes

Start_DAC starts the conversion of all register values into output voltages.

The low word of value (bits 0…15) is written to the odd numbered channel, the high word (bits 16…31) to the even numbered channel.

See also

Start_DAC, DAC

Valid for

AOut-4/16 Rev. B, AOut-4/16 Rev. C, 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:
  WriteDAC32(1,1,Data_1[i])'Set output register DAC1
  WriteDAC32(1,2,Data_2[i])'Set output register DAC2
  WriteDAC32(1,3,Data_3[i])'Set output register DAC3
  WriteDAC32(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
```

2.4 Pro I: Digital Modules

This section describes instructions, which apply to Pro I analog digital modules:

Counter

Digital I/Os

PWM outputs

SSI encoders

Comparator

Media / Storage

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

Cnt_Clear

`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. Bit = 0: No function. Bit = 1: Reset counter.	LONG

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 Pro-CO4-T Pro-CO4-I Pro-CO4-D	–	4	3	2	1

Notes

After processing `Cnt_Clear`, the reset counters start counting, and the bit `pattern` is reset to 0 (Null).

See also

Cnt_Enable, Cnt_SetMode

Valid for

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

Example

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

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

Init:
  Cnt_SetMode(module,1111b) 'Set counters 1…4 to
                            'clock/direction evaluation
  Cnt_Clear(module,1111b)   'Set values of counters 1…4 to 0
  Cnt_Enable(module,1111b)  'Enable counters 1…4
```

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

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 Pro-CO4-T Pro-CO4-I Pro-CO4-D		–		4	3	2	1

Notes

- / -

See also

Cnt_Clear, Cnt_SetMode

Valid for

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

Example

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

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

Init:
  Cnt_SetMode(module,1000b) 'Set counter 4 to clock/direction
                            'evaluation, all other counters to
                            '4 edge evaluation
  Cnt_Clear(module,1000b)   'Set counter values of counter 4 to 0
  Cnt_Enable(module,1000b)  'Enable counter 4, disable all others
```

Cnt_Latch

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

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 Pro-CO4-T Pro-CO4-I Pro-CO4-D		–		4	3	2	1

Notes

- / -

See also

Cnt_ReadLatch16, Cnt_ReadLatch32

Valid for

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

Example

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

```
#Include ADwinPro_All.Inc
#Define module 1
Dim value As Long

Init:
  value = ADC(module,1)     'Get measurement value
  If (value>49151) Then
    Cnt_Latch(module,0011b) 'Latch counters 1 and 2
    REM Calculate difference
    Par_1 = Cnt_ReadLatch32(module,1) - Cnt_ReadLatch32(module,2)
  EndIf
```

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 (1…16).	LONG
ret_val	Current counter value (16-bit value).	LONG

Notes

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

Valid for

CNT-16/16(-I)

Example

```
#Include ADwinPro_All.Inc
#Define module 1
Event:
  Par_1 = Cnt_Read16(module,1)'Get current value of counter 1
  Par_2 = Cnt_Read16(module,2)'Get current value of counter 2
```

Cnt_Read32

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 (1…4 / 1…8).	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.

Cnt_Latch	→	Cnt_ReadLatch32
Copy current count value to the latch register		Read out latch register

For special applications you can also use these instructions instead of **Cnt_Read32**.

See also

Cnt_Read16, Cnt_Latch, Cnt_ReadLatch32

Valid for

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

Example

```
#Include ADwinPro_All.Inc
#Define module 1
Event:
  Par_1 = Cnt_Read32(module,3)'Get current value of counter 3
  Par_2 = Cnt_Read32(module,4)'Get current value of counter 4
```

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 (1…16).	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

Valid for

CNT-16/16(-I)

Example

```
#Include ADwinPro_All.Inc
#Define module 1
Dim value As Long

Init:
  value = ADC(module,1)     'Get measurement value
  If (value > 49151) Then
    Cnt_Latch(module,0011b) 'Latch counters 1 and 2
    REM Difference counters 1 and 2
    Par_1 = Cnt_ReadLatch16(module,1) - Cnt_ReadLatch16(module,2)
  EndIf
```

Cnt_ReadLatch32

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 (1…4 / 1…8).	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

Valid for

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

Example

```
#Include ADwinPro_All.Inc
#Define module 1
Dim value As Long

Init:
  value = ADC(module,1)     'Get measurement value
  If (value > 49151) Then
    Cnt_Latch(module,0011b) 'Latch counters 1 and 2
    REM Difference counters 1 and 2
    Par_1 = Cnt_ReadLatch32(module,1) - Cnt_ReadLatch32(module,2)
  EndIf
```

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

module	Specified module address (1…255).	LONG
pattern	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 Pro-CNT-VR2PW2	4	3	2	1

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.

See also

Cnt_Clear, Cnt_Enable

Valid for

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

Example

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

Init:
  Cnt_SetMode(module,1100b) 'Counter 3 and 4 to clock/direction
                            'evaluation, all others to
                            'four edge evaluation
  Cnt_Clear(module,1100b)   'Set values of counters 3 and 4 to 0
  Cnt_Enable(module,1100b)  'Enable counters 3 and 4,
                            'disable all others
```

CO4_ClearEnable

CO4_ClearEnable enables the external input CLR of one or more counters.

Syntax

```
#Include ADwinPro_All.Inc
CO4_ClearEnable(module,pattern)
```

Parameters

module	Specified module address (1…255).	LONG
pattern	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".

Notes

The external input may be enabled either with CO4_ClearEnable or with CO4_LatchEnable, but not with both instructions simultaneously!

See also

CO4_LatchEnable

Valid for

CO4-D, CO4-I, CO4-T

Example

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

Init:
  CO4_SetMode(module,1,1)   'Counter 1: CLK/DIR mode
  CO4_LatchEnable(module,0)'Disable latch-input for all counters
  CO4_ClearEnable(module,1)'Enable clear-input for counter 1
  Cnt_Clear(module,1)       'Clear counter 1
  Cnt_Enable(module,1)      'Start counter 1
Event:
  Par_1 = CO4_Read(module,1)'Read out counter 1
```

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 (1…4).	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 (*Pro-CO4-D:* signal after the level converter)			**Status message** (Bit = 1) **Signal: Readable once**		**Error: Readable repeatedly**	
Input B	Input A	Input CLR/ LATCH	Signal LATCH exists	Signal CLR exists	Correla-tion error	Cable error

Error (repeatedly readable status message): Use `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 (< 0.33 V/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 `CO4_ClearEnable`).
Bit 3 = 1: LATCH signal exists (when it has been released by `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 `CO4_ResetStatus` has no effect.

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

See also

CO4_ResetStatus

Valid for

CO4-D, CO4-I, CO4-T

Example

```
#Include ADwinPro_All.Inc
#Define module 1
Dim error As Long

Init:
  CO4_SetMode(module,1,0)   'Counter 1 four edge evaluation mode
  Cnt_Clear(module,1)       'Clear counter 1
  Cnt_Enable(module,1)      'Start counter 1
  CO4_ResetStatus(module,1) 'Clear status register
  error = 0                 'Reset error code

Event:
  Par_1 = CO4_Read(module,1) 'Read out counter 1
  Par_2 = CO4_GetStatus(module,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(module,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

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

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 **CO4_ClearEnable** or with **CO4_LatchEnable**, but not with both instructions simultaneously!

See also

CO4_ClearEnable

Valid for

CO4-D, CO4-I, CO4-T

Example

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

Init:
  CO4_SetMode(module,1,1)   'Counter 1: CLK/DIR mode
  CO4_ClearEnable(module,0)'Disable clear-input for all counters
  CO4_LatchEnable(module,1)'Release latch-input (counter 1)
  Cnt_Clear(module,1)       'Clear counter 1
  Cnt_Enable(module,1)      'Start counter 1

Event:
  Par_1 = CO4_ReadLatch(module,1)'Read out counter 1
```

CO4_Read

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 (1…4 / 1…16).	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.

Cnt_Latch	→	CO4_ReadLatch
Copy current counter value to the latch register		Read out latch register

For specific applications you can also use these instructions instead of CO4_Read.

See also

Cnt_Latch, CO4_ReadLatch

Valid for

CNT-16/32(-I), CO4-D, CO4-I, CO4-T

Example

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

Init:
CO4_SetMode(module,1,1)    'Counter 1: CLK/DIR mode
Cnt_Clear(module,1)        'Clear counter 1
Cnt_Enable(module,1)       'Start counter 1

Event:
Par_1 = CO4_Read(module,1) 'Get current value of counter 1
```

CO4_ReadLatch

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 (1…4 / 1…16).	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 **Cnt_Latch** and can then be read by **CO4_ReadLatch**.

See also

Cnt_Latch, CO4_Read

Valid for

CNT-16/32(-I), CO4-D, CO4-I, CO4-T

Example

```
#Include ADwinPro_All.Inc
#Define module 1
Dim old, new As Long        'Dimensioning the variables

Init:
old = 0                     'Initialize the variable
CO4_SetMode(module,1,1)     'Counter 1: CLK/DIR mode
Cnt_Clear(module,1)         'Reset counter 1
Cnt_Enable(module,1)        'Start counter 1

Event:
Cnt_Latch(module,1)         'Latch counter 1 and…
new = CO4_ReadLatch(module,1)'read out latch
Par_1 = new - old           'Calculate difference
(f = impulses / time)
old = new                   'Save new counter value as old one
```

CO4_ResetStatus

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

module	Specified module address (1…255).	LONG
pattern	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 containts the status of a counter's input signals (read out using CO4_GetStatus).

See also

CO4_GetStatus

Valid for

CO4-D, CO4-I, CO4-T

Example

```
#Include ADwinPro_All.Inc
#Define module 1
Dim error As Long

Init:
  CO4_SetMode(module,1,0)   'Counter 1:four edge mode
  Cnt_Clear(module,1)       'Clear counter 1
  Cnt_Enable(module,1)      'Start counter 1
  CO4_ResetStatus(module,1) 'Clear status register
  error = 0                 'Reset error code

Event:
  Par_1 = CO4_Read(module,1) 'Read out counter 1
  Par_2 = CO4_GetStatus(module,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(module,1) 'Reset status register
    error = 0               'Reset error code
  EndIf
```

CO4_Set_Latch-Mode

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 tables); the default setting is 00000000b.	LONG

	Destination register for software latch (Cnt_Latch)				Copy latch content into additional latch			
Bit no. in pattern	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.

C04_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

Valid for

CO4-D, CO4-I, CO4-T

CO4_SetMode

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

module	Specified module address (1…255).	LONG
cnt_no	Counter number (1…4).	LONG
mode	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

Valid for

CO4-D, CO4-I, CO4-T

Example

```
#Include ADwinPro_All.Inc
#Define module 1
#Define refCLK 40E6
Dim rise, rise_old, fall, fall_old, T As Long

Init:
  rise_old = 0
  fall_old = 0
  CO4_SetMode(module,1,2)   'Counter 1: PWM analysis
  Cnt_Clear(module,1)       'Clear counter 1
  Cnt_Enable(module,1)      'Start counter 1

Event:
  rise = CO4_ReadLatch(module,1)'Read out latch 1
                            '(pos. edge, counter 1)
  fall = CO4_ReadLatch(module,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

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

For the modules Pro-DIO-32 and Pro-DIO-32 Rev. B, we recommend first programming the channels using the instructions **Digprog1** and **Digprog2** as inputs or outputs.

Depending on 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**, **SyncAll** has the same functions as **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

Valid for

DIO-32 Rev. A, 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

Example

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

Init:
Digprog1(module,0FFFFh)     'DIO15:00 (low-word) of DIO-32-
                            'module as output
Digprog2(module,0)          'DIO31:16 (high-word) of DIO-32-
                            'module as input
Dig_WriteLatch1(module,0)   'Set all output bits to 0

Event:
Dig_Latch(module)           'Latch inputs, output contents of the
                            'output latch
Rem more program steps
Par_1 = Dig_ReadLatch2(module)'Read in high-word and output at …
Dig_WriteLatch1(module,Par_1)'the next event in the low-word
```

Dig_ReadLatch1

Dig_ReadLatch 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

module	Specified module address (1…255).	LONG

Notes

We recommend first programming the specified channels as inputs using **Digprog1**.

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

- **Digin_Word**
- **Digin_Word2**
- **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

Valid for

DIO-32 Rev. A, DIO-32 Rev. B, OPT-16 Rev. A, OPT-16 Rev. B

Example

```
#Include ADwinPro_All.Inc
#Define module1 1
#Define module2 3
Dim value As Long

Init:
  REM Set DIO15:0 of modules 1+2 as inputs
  Digprog1(module1,0)
  Digprog1(module2,0)
  SyncEnable(module1,dio,1)'Enable synchronization of module 1
  SyncEnable(module2,dio,1)'Enable synchronization of module 2

Event:
  Rem Transfer the logic level at the digital inputs of both
  Rem modules synchronously to the latch register
  SyncAll()
  Par_1 = Dig_ReadLatch1(module1)'Read temp. register of module 1
  Par_2 = Dig_ReadLatch1(module2)'Read temp. register of module 2
```

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

module	Specified module address (1…255).	LONG

Notes

We recommend first programming the specified channels as inputs using **Digprog2**.

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

- **Digin_Word**
- **Digin_Word2**
- **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

Valid for

DIO-32 Rev. A, DIO-32 Rev. B

Example

```
#Include ADwinPro_All.Inc
#Define module1 3
#Define module2 5
Dim value As Long

Init:
  REM Set DIO31:16 of modules 1+2 as inputs
  Digprog2(module1,0)
  Digprog2(module2,0)
  SyncEnable(module1,dio,1)'Enable synchronization of module 1
  SyncEnable(module2,dio,1)'Enable synchronization of module 2

Event:
Rem Transfer the logic level at the digital inputs of both modules
Rem synchronously to the latch register
  SyncAll()
  Par_1 = Dig_ReadLatch2(module1)'Read temp. register of module 1
  Par_2 = Dig_ReadLatch2(module2)'Read temp. register of module 2
```

Dig_WriteLatch1

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 **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 **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

Valid for

DIO-32 Rev. A, 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
#Define adc_module 2
#Define d_module1 3
#Define d_module2 5
Dim value As Long

Init:
  REM DIO-32 only: Set DIO15:00 of the module as outputs
  Digprog1(d_module1,0FFFFh)
  Digprog1(d_module2,0FFFFh)

  SyncEnable(d_module1,dio,1)'Enable synchronization of digital
                             'module 1
  SyncEnable(d_module2,dio,1)'… and digital module 2
  Dig_WriteLatch1(d_module1,1)'Set lowest bit in the output
                             'latch register
  Dig_WriteLatch1(d_module2,1)'Set lowest bit in the output
                             'latch register

Event:
  value = ADC(adc_module,1)'Measurement data acquisition
  If (value > 3000) Then   'Limit value exceeded?
    SyncAll()              'Output values from the latch
                           'registers
  EndIf
```

Dig_WriteLatch2

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 (see table).	LONG

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 **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 instructionin 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 **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

Valid for

DIO-32 Rev. A, DIO-32 Rev. B

Example

```
#Include ADwinPro_All.Inc
#Define adc_module 2
#Define d_module1 3
#Define d_module2 5
Dim value As Long

Init:
  REM DIO-32 only: Set DIO31:16 of the module as outputs
  Digprog2(d_module1,0FFFFh)
  Digprog2(d_module2,0FFFFh)

  SyncEnable(d_module1,dio,1)'Enable synchronization of module 1
  SyncEnable(d_module2,dio,1)'… and module 2
  Dig_WriteLatch2(d_module1,1)'Set bit 16 in output latch
  Dig_WriteLatch2(d_module2,10000b)'Set bit 20 in output latch

Event:
  value = ADC(adc_module,1)'Measurement data acquisition
  If (value > 3000) Then    'Limit value exceeded?
    SyncAll()               'Output values from the latch
                            'registers
  EndIf
```

Dig_WriteLatch32

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

Valid for

DIO-32 Rev. A, DIO-32 Rev. B

Example

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

Init:
  Digprog1(module,0FFFFh)   'DIO15:00 (low-word) of DIO-32-
                            'module as output
  Digprog2(module,0FFFFh)   'DIO31:16 (high-word) of DIO-32-
                            'module as output

Event:
  Rem Output information of the output latch on a DIO-32 board
  Dig_Latch(module)
  Dig_WriteLatch32(module,Par_1)'Write long-word to output latch
```

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

module	Specified module address (1…255).	LONG
ret_val	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

We recommend first programming 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

Valid for

DIO-32 Rev. B

Example

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

Init:
  Digprog1(module,0)        'Set DIO15:00 (Low-Word) as inputs
  Digprog2(module,0)        'Set DIO31:16 (High-Word) as inputs

Event:
  Par_1 = Digin_Long_F(module)'Read all inputs
```

Digin_Word1

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_Word(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: 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

For the modules Pro-DIO-32 and Pro-DIO-32 Rev. B, we recommend first programming the channels as inputs using the instructions **Digprog1** and **Digprog2**.

See also

Digin_Word2, Digout, Digout_Word1, Digout_Word2, Digprog1, Digprog2

Valid for

DIO-32 Rev. A, DIO-32 Rev. B, OPT-16 Rev. A, OPT-16 Rev. B

Example

```
#Include ADwinPro_All.Inc
#Define adc_module 1
#Define dio_module 4
Dim Data_1[10000] As Long As FIFO

Init:
  REM DIO32 only: Set DIO15:00 (Low-Word) as inputs
  Digprog1(dio_module,0)

Event:
  If (Digin_Word(dio_module) And 14 = 14) Then
                            'Query, if inputs 1, 2 and 3
                            'on mdoule 4 are set
    Data_1 = ADC(adc_module,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_Word(dio_module) And 1110b = 1110b) Then
  Data_1 = ADC(adc_module,1) 'Measurement data acquisition
EndIf
```

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

We recommend first programming the specified channels as inputs using **Digprog2**.

See also

Digin_Word1, Digout, Digout_Word1, Digout_Word2, Digprog1, Digprog2

Valid for

DIO-32 Rev. A, DIO-32 Rev. B

Example

```
#Include ADwinPro_All.Inc
#Define adc_module 1
#Define dio_module 4
Dim Data_1[10000] As Long As FIFO

Init:
  Digprog2(dio_module,0)    'DIO31:16 (high word) as inputs

Event:
   'Query if inputs 16, 18 and 21 are set on module 4
  If (Digin_Word2(dio_module) And 39 = 39) Then
    Data_1 = ADC(adc_module,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(dio_module) And 0100101b = 0100101b) Then
  Data_1 = ADC(adc_module,1)'Measurement data acquisition
EndIf
```

Digout

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 using **Digout_F**, because it works faster and needs essentially less program memory.

The specified channel must be first programmed as output using **Digprog1** or **Digprog2**.

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

Get_Digout_Word1/2	→	…	→	Digout_Word1/2
Read current status of all digital outputs		Set or reset specified 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

Valid for

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
#Define module 1
Dim value As Long

Init:
  REM DIO32 only: Set channels as outputs
  Digprog1(module,0FFFFh)   'DIO15:00 (low word) as outputs
  Digprog2(module,0FFFFh)   'DIO31:16 (high word) as outputs

Event:
  value = ADC(module,1)     'Measurement data acquisition
  If (value < 100) Then     'If value is below limit
    Digout(module,2,0)      'reset digital output 2
  EndIf
```

Digout_Bits_F

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

module	Specified module address (1…255).	LONG
set	Bit pattern that sets specified digital outputs to the level "high". Bit = 0: Output remains unchanged. Bit = 1: Set output to level "high".	LONG
clear	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 **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

Valid for

DIO-32 Rev. B, TRA-16 Rev. B

Example

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

Init:
  REM DIO32 only: Set channels as outputs
  Digprog1(module,0FFFFh)   'DIO15:00 (low word) as outputs
  Digprog2(module,0FFFFh)   'DIO31:16 (high word) as outputs

Event:
  If (Par_1 = 1) Then       'Get condition
    REM lower word: Set byte MSBs, clear all other bits
    Digout_Bits_F(module,8080h,7F7Fh)
  Else
    Rem lower word: Set odd-numbered bits, clear even-numbered
    Digout_Bits_F(module,5555h,0AAAAh)
  EndIf
```

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 **Digout**, that works on software level.

See also

Digout, Digout_Bits_F, Digout_Word1, Digout_Word2, Digprog1, Digprog2

Valid for

DIO-32 Rev. B, TRA-16 Rev. B

Example

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

Init:
  REM DIO32 only: Set channels as outputs
  Digprog1(module,0FFFFh)   'DIO15:00 (low word) as outputs
  Digprog2(module,0FFFFh)   'DIO31:16 (high word) as outputs

Event:
  If (Digin_Word(module) And 8000h = 8000h) Then
                            'Read low-word (bits 0…15) and
                            'check, if MSB is set
    Digout_F(module,31,0)   'If MSB is set, clear bit 31
  Else
    Digout_F(module,31,1)   'If MSB is cleared, set bit 31
  EndIf
```

Digout_Long_F

With the given 32-bit value, **Digout_Long_F** sets or clears all outputs on the specified module.

Syntax

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

Parameters

module	Specified module address (1…255).	LONG
pattern	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

Valid for

DIO-32 Rev. B

Example

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

Init:
  Digprog1(module,0FFFFh)   'DIO15:00 (low word) as outputs
  Digprog2(module,0FFFFh)   'DIO31:16 (high word) as outputs

Event:
  Digout_Long_F(module,1000000)'Output the value 1 million as
                               'binary value on DIOs
```

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

module	Specified module address (1…255).	LONG
pattern	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 **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_Word

Valid for

DIO-32 Rev. A, 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
#Define adc_module 1
#Define dio_module 4
Dim value As Long

Init:
  REM DIO32 only: Set channels as outputs
  Digprog1(dio_module,0FFFFh)'DIO15:00 (low word) as outputs

Event:
  value = ADC(adc_module,1)'Measurement data acquisition
  If (value > 3000) Then    'Is limit value exceeded?
    Digout_Word1(dio_module,111b)'Set outputs 0, 1 and 2 of the
                            'DIO module, other outputs are reset
  EndIf
```

Digout_Word2

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

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

Valid for

DIO-32 Rev. A, DIO-32 Rev. B

Example

```
#Include ADwinPro_All.Inc
#Define adc_module 1
#Define dio_module 4
Dim value As Long

Init:
  Digprog2(dio_module,0FFFFh)'DIO31:16 (high word) as outputs

Event:
  value = ADC(adc_module,1)'Measurement data acquisition
  If (value > 3000) Then    'Is limit value exceeded?
    Digout_Word2(dio_module,1011b)'set outputs 16, 17 a. 19, all
                            'other outputs are reset
  EndIf
```

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

module	Specified module address (1…255).	LONG
pattern	Bit pattern that sets the channels as inputs or outputs: Bit = 0: Set channel as input. Bit = 1: Set channel as output.	LONG

Module	Bit no.	31…16	15	…	8	7	…	0
DIO-32 Rev. A	channel no.	–	15	…	08	07	…	00
DIO-32 Rev. B	channel no.	–	–	–	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

Valid for

DIO-32 Rev. A, DIO-32 Rev. B

Example

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

Init:
  Digprog1(module, 11111111b)'Configures channels 0…7 of DIO
                             'module no. 1 as outputs and channels
                             '8…15 as inputs
```

Digprog2

Digprog2 programs all channels 16…31 of the specified module as inputs or outputs.

Syntax

```
#Include ADwinPro_All.Inc
Digprog2(module,pattern)
```

Parameters

module	Specified module address (1…255).	LONG
pattern	Bit pattern that sets the channels as inputs or outputs: Bit = 0: Set channel as input. Bit = 1: Set channel as output.	LONG

Module	Bit no.	31…16	15	…	8	7	…	0
DIO-32 Rev. A	channel no.	–	31	…	24	23	…	16
DIO-32 Rev. B	channel no.	–	–	–	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

Valid for

DIO-32 Rev. A, DIO-32 Rev. B

Example

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

Init:
  Digprog2(module, 11111111b)'Configures channels 16…23 of the
                             'DIO module no. 1 as outputs and
                             ' channels 24…31 as inputs
```

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

module	Specified module address (1…255).	LONG
cnt_select	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

- / -

See also

Cnt_Clear, Cnt_Enable, Cnt_Latch, Cnt_Read32, Cnt_ReadLatch32, Cnt_SetMode

Valid for

CNT-VR4L(-I)

Example

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

Init:
  REM Enable latch-inputs of counters 1+2,
  REM disable latch-inputs of counters 3+4
  ExtLch_Enable(module, 0011b)
```

Get_Digout_Long

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

Valid for

DIO-32 Rev. B

Example

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

Event:
  Par_1 = Get_Digout_Long(module)'return bits 31:00 from latch
```

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

Valid for

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
#Define module 1
Dim value As Long

Init:
  value = Get_Digout_Word1(module)'Read current value of the
                                  'output register
  value = value And 0FFFEh 'Set LSB (bit 0) to 0
  Digout_Word1(module,value)'Return changed value
```

Get_Digout_Word2

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

Valid for

DIO-32 Rev. B

Example

```
#Include ADwinPro_All.Inc
#Define module 1
Dim value As Long

Init:
  value = Get_Digout_Word2(module)'Read current value of the
                                  'output register
  value = value And 0FFFDh 'Set bit 17 to 0
  Digout_Word2(module,value)'Return changed value
```

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

module	Specified module address (1…255).	LONG
output	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

The instruction *does not* change the level of the PWM outputs.
If you want to change the level of the PWM outputs, use **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

Valid for

PWM-4(-I)

Example

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

Init:
  PWM_Enable(module,1)      'Enable PWM output 1 for output
  PWM_Set(module,1,0,2500,2500)'Set PWM signal for output 1
```

PWM_Out

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

module	Specified module address (1…255).	LONG
channel	Number of the PWM output channel (1…4).	LONG
level	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

Valid for

PWM-4(-I)

Example

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

Init:
  PWM_Enable(module,11b)    'Enable counters of the PWM outputs
                            '1 and 2
  PWM_Set(module,1,0,2500,2500)'Set PWM signal for output 1
  PWM_Out(module,2,1)       'Set PWM output 2 to logical value 1
```

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:

$$f_{out} = \frac{5\,\mathrm{MHz}}{2^{prescale} \cdot (low + high)}$$

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 smallest output frequency at a still definable duty cycle of approx. 0...100% is about 0.6 Hz.

The highest output frequency where the duty cycle can be still defined in 1%-steps, is 50 kHz.

See also

PWM_Enable, PWM_Out

Valid for

PWM-4(-I)

Example

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

Init:
  PWM_Enable(module,11b)    'Enable PWM outputs 1 and 2 for
                            'output
  PWM_Set(module,1,0,2500,2500)'Set PWM signal for output 1
  PWM_Out(module,2,1)       'Set PWM output 2 to logical value "1"
```

SSI_Mode

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

module	Specified module address (1…255).	LONG
pattern	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. **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 corrct 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

Valid for

CO4-D

Example

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

Init:
  SSI_Set_Clock(module,200)'CLK (clock rate) = 50 kHz
  SSI_Mode(module,3)       'Set continuous-mode
                           '(for both encoders)
  SSI_Set_Bits(module,1,23)'Amount of encoder bits=23 (encoder 1)
  SSI_Set_Bits(module,2,23)'Amount of encoder bits=23 (encoder 2)

Event:
  Par_1 = SSI_Read(module,1)'Read out and display position value
                            '(encoder 1)
  Par_2 = SSI_Read(module,2)'Read out and display position value
                            '(encoder 2)
```

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,dcdr_no)
```

Parameters

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

Valid for

CO4-D

Example

```
#Include ADwinPro_All.Inc
#Define module 1
Dim m, n, y As Long

Init:
  SSI_Set_Clock(module,50)  'CLK (clock rate) = 200 kHz
  SSI_Mode(module,1)        'Set continuous-mode (encoder 1)
  SSI_Set_Bits(module,1,23) 'Amount of encoder bits=23 (encoder 1)

Event:
  Par_1 = SSI_Read(module,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

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,dcdr_no,bit_no)
```

Parameters

module	Specified module address (1…255).	LONG
dcdr_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

Valid for

CO4-D

Example

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

Init:
  SSI_Set_Clock(module,50)  'CLK (clock rate) = 200 kHz
  SSI_Mode(module,3)        'Set continuous-mode (for both
                            'encoders)
  SSI_Set_Bits(module,1,10) '10 encoder bits for encoder 1
  SSI_Set_Bits(module,2,25) '25 encoder bits for encoder 2

Event:
  Par_1 = SSI_Read(module,1) 'Read out and display position value
                             '(encoder 1)
  Par_2 = SSI_Read(module,2) 'Read out and display position value
                             '(encoder 2)
```

SSI_Set_Clock

SSI_Set_Clock sets the clock rate (approx. 40 kHz to 1 MHz) on the specified module, with which the encoder is clocked.

Syntax

```
#Include ADwinPro_All.Inc
SSI_Set_Clock(module,prescale)
```

Parameters

module	Specified module address (1…255).	LONG
prescale	scale factor (10…255) for setting the clock rate according to the equation: Clock rate = 10 MHz / prescale.	LONG

Notes

The setting of the clock rate is always identical for both encoders, which are connected to the module, and cannot be set separately. If necessary, the clock has to consider the clock rate of the slowest encoder.

After start-up of the module the default scale factor of 128 is used, corresponding to 78 kHz.

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

Valid for

CO4-D

Example

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

Init:
  SSI_Set_Clock(module,10)  'CLK (clock rate) = 1 MHz
  SSI_Mode(module,3)        'Set continuous-mode
                            '(for both encoders)
  SSI_Set_Bits(module,1,10) 'Amount of encoder bits=10 (encoder 1)
  SSI_Set_Bits(module,2,25) 'Amount of encoder bits=25 (encoder 2)

Event:
  Par_1 = SSI_Read(module,1) 'Read out and display position value
                             '(encoder 1)
  Par_2 = SSI_Read(module,2) 'Read out and display position value
                             '(encoder 2)
```

SSI_Start

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,pattern)
```

Parameters

module	Specified module address (1…255).	LONG
pattern	Bit pattern for selecting the SSI decoders, which are to be started: Bit = 0: No function. Bit = 1: Start reading of the SSI decoder.	LONG

Bit no.	31:2	1	0
SSI decoder	–	2	1

Notes

In 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

Valid for

CO4-D

Example

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

Init:
  SSI_Set_Clock(module,250)'CLK (clock rate) = approx. 40 kHz
  SSI_Mode(module,0)       'Set single shot-mode
                           '(both counters)
  SSI_Set_Bits(module,1,23)'Amount of encoder bits=23 (encoder 1)
  SSI_Set_Bits(module,2,23)'Amount of encoder bits=23 (encoder 2)

Event:
  SSI_Start(module,3)      'Read position value of encoders 1 & 2
  Do                       'for encoder 1:
  Until (SSI_Status(module,1) = 0)
  REM If position value is read completely, then …
  Par_1 = SSI_Read(module,1)'read out and display position value
  Do                       'For encoder 2:
  Until (SSI_Status(module,2) = 0)
  REM If position value is read completely, then …
  Par_1 = SSI_Read(module,2)'read out and display position value
```

SSI_Status

SSI_Status returns the current read-status on the speicified module for a specified decoder.

Syntax

```
#Include ADwinPro_All.Inc
ret_val = SSI_Status(module,dcdr_no)
```

Parameters

module	Specified module address (1…255).	LONG
dcdr_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 was has been read. 1: Decoder is reading an encoder value.	LONG

Notes

Use the status query only in the SSI mode "single shot". In "continuous" mode, querying the status is not useful.

See also

SSI_Mode, SSI_Read, SSI_Set_Bits, SSI_Set_Clock, SSI_Start

Valid for

CO4-D

Example

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

Init:
  SSI_Set_Clock(module,250)'CLK (clock rate) = approx. 40 kHz
  SSI_Mode(module,0)       'Set single shot-mode
                           '(both counters)
  SSI_Set_Bits(module,1,23)'Amount of encoder bits=23 (encoder 1)
  SSI_Set_Bits(module,2,23)'Amount of encoder bits=23 (encoder 2)

Event:
  SSI_Start(module,3)      'Read position value of encoders 1 & 2
  Do                       'For encoder 1:
  Until (SSI_Status(module,1) = 0)
  REM If position value is read completely, then …
  Par_1 = SSI_Read(module,1)'Read out and display position value
  Do                       'For encoder 2:
  Until (SSI_Status(module,2) = 0)
  REM If position value is read completely, then …
  Par_1 = SSI_Read(module,2)'Read out and display position value
```

Comp_Digin_Word

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

module	Specified module address (1…255).	LONG
ret_val	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

Valid for

COMP-16 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_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

module	Specified module address (1…255).	LONG
ret_val	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 $value_1$ - $value_2$, $value_3$ - $value_4$, …, $value_{15}$ - $value_{16}$. 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

Valid for

COMP-16 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

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.	LONG
array2[]	Destination array for 1024 measurement values of the channel with the higher channel number.	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 use d for the modules

Pro-COMP-16 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_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

Valid for

COMP-16 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

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

Conversion digits into Volt

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:

$$\text{voltage} = \text{digits} \cdot \frac{V_{max} - V_{min}}{1024} + V_{min}$$

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

Valid for

COMP-16 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_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

module	Specified module address (1…255).	LONG
pattern	Bit pattern for resetting the minimum and maximum values (For the assigment 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

Valid for

COMP-16 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

`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

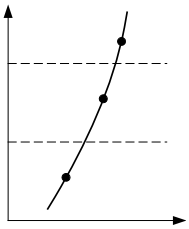
`module`	Specified module address (1…255).	LONG
`channel`	Channel number(1…16).	LONG
`ValHigh`	Upper threshold value (1…1023) of the channel.	LONG
`ValLow`	Lower threshold value (0…1022) of the channel.	LONG

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. (see `Comp_Read` for conversion of digit into volt).

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:



Meas. value > upper threshold	→ Set bit
Meas. value is equal to or between the threshold values	→ Keep bit unchanged
Meas. value < lower threshold	→ Delete bit

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 `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 `Comp_Digin_Word_Diff`.

See also

Comp_Read, Comp_Reset, Comp_Fifo_Select, Comp_Digin_Word, Comp_Digin_Word_Diff, Comp_Fifo_Read

Valid for

COMP-16 Rev. A

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

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

Valid for

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_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

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

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

Valid for

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

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

module	Specified module address (1…255).	LONG
f_info[]	Array, which contains the numbers of the start and end sectors of all files.	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.	LONG
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_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1

If bit = 1 (bit = 0: without meaning):
A_1 A storage medium is in the slot.
A_2 Unknown error occured.
A_3 Glue-logic is busy.
A_4 Data can be read.
A_5 Data is written.
A_6 Reset is just being executed.
A_7 End of file exceeded, no data is transferred.
A_8 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.

The array `array[]` must at least contain `array_idx`+`s_cnt`×128 values.

See also

Media_RD_Blk_L, Media_RD_Blk_F

Valid for

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: 29x128
#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

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.	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.	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_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1

If bit = 1 (bit = 0: without meaning):
A_1 A storage medium is in the slot.
A_2 Unknown error occured.
A_3 Glue-logic is busy.
A_4 Data can be read.
A_5 Data is written.
A_6 Reset is just being executed.
A_7 End of file exceeded, no data is transferred.
A_8 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-

tor 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

Valid for

Storage Rev. A

Media_RD_Blk_L

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

module	Specified module address (1…255).	LONG
f_info[]	Array, which contains the numbers of the start and end sectors of all files.	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.	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_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1

If bit = 1 (bit = 0: without meaning):
A_1 A storage medium is in the slot.
A_2 Unknown error occured.
A_3 Glue-logic is busy.
A_4 Data can be read.
A_5 Data is written.
A_6 Reset is just being executed.
A_7 End of file exceeded, no data is transferred.
A_8 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+s_cnt×128 array elements.

See also

Media_WR_Blk_L, Media_RD_Blk_F

Valid for

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
REM the array length with the sine data must be a multiple
REM of 128 and greater or equal "nds":
#Define lng 3712               'here: 29x128
#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, nds 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

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.	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.	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_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1

If bit = 1 (bit = 0: without meaning):
A_1 A storage medium is in the slot.
A_2 Unknown error occured.
A_3 Glue-logic is busy.
A_4 Data can be read.
A_5 Data is written.
A_6 Reset is just being executed.
A_7 End of file exceeded, no data is transferred.
A_8 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.

See also

Media_WR_Blk_L, Media_RD_Blk_L

Valid for

Storage Rev. A

Media_RD_FileInfo

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.	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_6	A_5	A_4	A_3	A_2	A_1

If bit = 1 (bit = 0: without meaning):
A_1 A storage medium is in the slot.
A_2 Unknown error occured.
A_3 Glue-logic is busy.
A_4 Data can be read.
A_5 Data is written.
A_6 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_RD_Blk_L, Media_RD_Blk_F

Valid for

Storage Rev. A

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
```

2.5 Pro I: Signal Conditioning and Interface Modules

This section describes instructions, which apply to Pro I signal conditioning and interface modules.

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

The instructions are sorted by type of interface:

- Thermocouples, page 168
- CAN bus, page 184
- Field bus, page 204
- RSxxx, page 216
- LS-Bus + Pro I, page 478

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

2.5.1 Thermocouples

This section describes instructions, which apply to Pro I thermocouple modules:

- PT100_Dig_To_Temp (page 169)
- PT100_Dig_To_R (page 170)
- TC_Select (page 171)
- TCJ_Dig_To_Temp (page 173)
- TCK_Dig_To_Temp (page 174)
- TC_Read_B (page 175)
- TC_Read_E (page 176)
- TC_Read_J (page 177)
- TC_Read_K (page 178)
- TC_Read_N (page 179)
- TC_Read_R (page 180)
- TC_Read_S (page 181)
- TC_Read_T (page 182)
- TC_Set_Rate (page 183)

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 or 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: Resistance Thermometer 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

Valid for

- / -

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
    Sleep(50) '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

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:

$$\text{ret\_val} = 100\,\Omega + 200\,\Omega \cdot \frac{\text{dig\_val} - 32768}{65536}$$

See also

TC_Select, PT100_Dig_To_Temp

Valid for

PT100-4, PT100-8

Example

It is presumed in the example that the analog output of the resistance thermometer 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
    Sleep(50) 'wait for settling time
    REM read thermoelectric voltage and convert into °C
    temp[channel] = PT100_Dig_To_R(ADC(1,1))
  Next
```

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

Valid for

- / -

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

Valid for

- / -

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
    Sleep(50)                 '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
  Rem …
  FPar_8 = temp[8]            'Temperature channel 8 in °C
```

TCK_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.

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 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 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

Valid for

- / -

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
    Sleep(50)                 '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
  Rem …
  FPar_8 = temp[8]            'Temperature channel 8 in °C
```

TC_Read_B

TC_Read_B returns the thermoelectric voltage (µV) or the temperature (°C / °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

module	Specified module address (1…255).	LONG
channel	Channel number (1…8).	LONG
ret_type	Type of return value ret_val: 0: Thermoelectric voltage in µV. 1: Temperature in °C. 2: Temperature in °F.	LONG
ret_val	Measured value of the channel, refers to ret_type: Thermoelectric voltage: 291 µV … 13820 µV. Temperature in °C: 250°C … 1820°C. Temperature in °F: 482°F … 3329.6°F.	FLOAT

Notes

The module samples the temperature regularly (setting of the sample rate see **TC_Set_Rate**). **TC_Read_B** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in °C and °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

Valid for

TC-8-ISO Rev. E

Example

```
#Include ADwinPro_All.Inc

Event:
  REM Read temperature in °C from channel 5
  FPar_1 = TC_Read_B(1,5,1)
```

TC_Read_E

TC_Read_E returns the thermoelectric voltage (µV) or the temperature (°C / °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 ret_val: 0: Thermoelectric voltage in µV. 1: Temperature in °C. 2: Temperature in °F.	LONG
ret_val	Measured value of the channel, refers to ret_type: Thermoelectric voltage: -8825µV … 76373µV. Temperature in °C: -200°C … 1000°C. Temperature in °F: -328°F … 1832°F.	FLOAT

Notes

The module samples the temperature regularly (setting of the sample rate see **TC_Set_Rate**). **TC_Read_E** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in °C and °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

Valid for

TC-8-ISO Rev. E

Example

```
#Include ADwinPro_All.Inc

Event:
  REM Read temperature in °C from channel 5
  FPar_1 = TC_Read_E(1,5,1)
```

TC_Read_J

TC_Read_J returns the thermoelectric voltage (µV) or the temperature (°C / °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

module	Specified module address (1…255).	LONG
channel	Channel number (1…8).	LONG
ret_type	Type of return value ret_val: 0: Thermoelectric voltage in µV. 1: Temperature in °C. 2: Temperature in °F.	LONG
ret_val	Measured value of the channel, refers to ret_type: Thermoelectric voltage: -8095µV … 69553µV. Temperature in °C: -210°C … 1200°C. Temperature in °F: -346°F … 2192°F.	FLOAT

Notes

The module samples the temperature regularly (setting of the sample rate see **TC_Set_Rate**). **TC_Read_J** returns the last sampled temperature value.

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.

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

Valid for

TC-8-ISO Rev. E

Example

```
#Include ADwinPro_All.Inc

Event:
  REM Read temperature in °C from channel 5
  FPar_1 = TC_Read_J(1,5,1)
```

TC_Read_K

TC_Read_K returns the thermoelectric voltage (µV) or the temperature (°C / °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 ret_val: 0: Thermoelectric voltage in µV. 1: Temperature in °C. 2: Temperature in °F.	LONG
ret_val	Measured value of the channel, refers to ret_type: Thermoelectric voltage: -5891 µV … 54886 µV. Temperature in °C: -200°C … 1372°C. Temperature in °F: -328°F … 2501.6°F.	FLOAT

Notes

The module samples the temperature regularly (setting of the sample rate see TC_Set_Rate). TC_Read_K returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in °C and °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

Valid for

TC-8-ISO Rev. E

Example

```
#Include ADwinPro_All.Inc

Event:
  REM Read temperature in °C from channel 5
  FPar_1 = TC_Read_K(1,5,1)
```

TC_Read_N

TC_Read_N returns the thermoelectric voltage (µV) or the temperature (°C / °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 ret_val: 0: Thermoelectric voltage in µV. 1: Temperature in °C. 2: Temperature in °F.	LONG
ret_val	Measured value of the channel, refers to ret_type: Thermoelectric voltage: -3990µV … 47513µV. Temperature in °C: -200°C … 1300°C. Temperature in °F: -328°F … 2372°F.	FLOAT

Notes

The module samples the temperature regularly (setting of the sample rate see **TC_Set_Rate**). **TC_Read_N** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in °C and °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

Valid for

TC-8-ISO Rev. E

Example

```
#Include ADwinPro_All.Inc

Event:
  REM Read temperature in °C from channel 5
  FPar_1 = TC_Read_N(1,5,1)
```

TC_Read_R

TC_Read_R returns the thermoelectric voltage (µV) or the temperature (°C / °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 ret_val: 0: Thermoelectric voltage in µV. 1: Temperature in °C. 2: Temperature in °F.	LONG
ret_val	Measured value of the channel, refers to ret_type: Thermoelectric voltage: -226 µV … 21101 µV. Temperature in °C: -50°C … 1768°C. Temperature in °F: -58°F … 3214.4°F.	FLOAT

Notes

The module samples the temperature regularly (setting of the sample rate see TC_Set_Rate). TC_Read_R returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in °C and °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

Valid for

TC-8-ISO Rev. E

Example

```
#Include ADwinPro_All.Inc

Event:
  REM Read temperature in °C from channel 5
  FPar_1 = TC_Read_R(1,5,1)
```

TC_Read_S

TC_Read_S returns the thermoelectric voltage (µV) or the temperature (°C / °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

module	Specified module address (1…255).	LONG
channel	Channel number (1…8).	LONG
ret_type	Type of return value ret_val: 0: Thermoelectric voltage in µV. 1: Temperature in °C. 2: Temperature in °F.	LONG
ret_val	Measured value of the channel, refers to ret_type: Thermoelectric voltage: -236 µV … 18693 µV. Temperature in °C: -50°C … 1768°C. Temperature in °F: -58°F … 3214.4°F.	FLOAT

Notes

The module samples the temperature regularly (setting of the sample rate see **TC_Set_Rate**). **TC_Read_S** returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in °C and °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

Valid for

TC-8-ISO Rev. E

Example

```
#Include ADwinPro_All.Inc

Event:
  REM Read temperature in °C from channel 5
  FPar_1 = TC_Read_S(1,5,1)
```

TC_Read_T

TC_Read_T returns the thermoelectric voltage (µV) or the temperature (°C / °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 ret_val: 0: Thermoelectric voltage in µV. 1: Temperature in °C. 2: Temperature in °F.	LONG
ret_val	Measured value of the channel, refers to ret_type: Thermoelectric voltage: -5603µV … 20872µV. Temperature in °C: -200°C … 400°C. Temperature in °F: -454°F … 752°F.	FLOAT

Notes

The module samples the temperature regularly (setting of the sample rate see TC_Set_Rate). TC_Read_T returns the last sampled temperature value.

The thermoelectric voltage and the temperature values in °C and °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

Valid for

TC-8-ISO Rev. E

Example

```
#Include ADwinPro_All.Inc

Event:
  REM Read temperature in °C from channel 5
  FPar_1 = TC_Read_T(1,5,1)
```

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

module	Specified module address (1…255).	LONG
rate	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

Valid for

TC-8-ISO Rev. E

Example

```
#Include ADwinPro_All.Inc

Init:
  REM Set sampling rate to 27.5 Hz
  TC_Set_Rate(1,8)
```

2.5.2 CAN bus

This section describes instructions, which apply to Pro I CAN bus modules:

CAN_Msg

CAN_Msg is a one-dimensional array consisting of 9 elements, where the message objects of the CAN bus are saved during sending and receiving.

Syntax

```
#Include ADwinPro_All.Inc
CAN_Msg[n] = value
```

or

```
ret_val = CAN_Msg[n]
```

Parameters

n	Element number in the array **CAN_Msg** (1… 9).	LONG
value	Value (8 bit), which is written into the message object.	LONG
ret_val	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 **Transmit**.

See also

En_Receive, En_Transmit, Read_Msg, Transmit

Valid for

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
```

En_Interrupt

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,msg_no)
```

Parameters

module	Specified module address (1…255).	LONG
channel	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
msg_no	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

Valid for

CAN-1, CAN-2

Example

```
#Include ADwinPro_All.Inc
Init:
  REM Initialize CAN controller 1 on 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_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,msg_no,id,id_extend)
```

Parameters

module	Specified module address (1…255).	LONG
channel	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
msg_no	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^{11} or 0…2^{29}).	LONG
id_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

Valid for

CAN-1, CAN-2

Example

```
#Include ADwinPro_All.Inc

Init:
  REM Initialize CAN controller 1 on 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

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,msg_no,id,id_extend)
```

Parameters

module	Specified module address (1…255).	LONG
channel	Number (1, 2) of the CAN channel that specifies the CAN controller.	LONG
msg_no	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^{11} or 0…2^{29}).	LONG
id_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, Transmit_Status

Valid for

CAN-1, CAN-2

Example

```
#Include ADwinPro_All.Inc
Init:
  REM Initialize CAN controller 1 on 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)
```

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 numbers of the CAN controller are found in the data-sheet AN82527 from Intel® (address map), e.g.:

- address 00h: controll register
- address 01h: status register
- address 5fh: interrupt register

See also

En_Interrupt, Init_CAN, Set_CAN_Baudrate, Set_CAN_Reg

Valid for

CAN-1, CAN-2

Example

```
#Include ADwinPro_All.Inc
Init:
  REM Initialize CAN controller 1 on 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

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

module	Specified module address (1…255).	LONG
channel	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 125 kBit/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

Valid for

CAN-1, CAN-2

Example

```
#Include ADwinPro_All.Inc
Init:
  REM Initialize CAN controller 1 on CAN module 1
  Init_CAN(1,1)
```

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,msg_no)
```

Parameters

module	Specified module address (1…255).	LONG
channel	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
msg_no	Number (1… 15) of the message object in the CAN controller.	LONG
ret_val	>0: A new message has arrived, the value is the identifier of the message object. -1: No new message has arrived.	LONG

Notes

To receive a message you have to follow the correct order:

- Once: Enable the message object with **En_Receive** for receiving.
- As often as needed: Check for a received message and save to **CAN_Msg** with **Read_Msg**.

You can read a received message only once.

See also

CAN_Msg, En_Receive, Init_CAN, Read_Msg_Con

Valid for

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
```

Read_Msg_Con

Read_Msg_Con returns if a completely new message has been received in a message object of one of the CAN controllers on the module.

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_Con(module,can_no,msg_no)
```

Parameters

module	Specified module address (1…255).	LONG
can_no	Number (1, 2) of the CAN channel that determines the CAN controller.	LONG
msg_no	Number (1… 15) of the message object in the CAN controller.	LONG
ret_val	>0: A new message has arrived, the value is the identifier of the message object. -1: No new message has arrived.	LONG

Notes

In contrary to **Read_Msg**, **Read_Msg_Con** makes sure the message is consistent: If a new message arrives while reading an old message, there is no mixture of old and new message.

To receive a message, follow these steps:

- Enable the message object for receive with **En_Receive**.
- Check for a new message, and if, store the message in **CAN_Msg** with **Read_Msg**.

You can read a received message only once.

See also

CAN_Msg, En_Interrupt, En_Receive, En_Transmit, Read_Msg

Valid for

CAN-1, CAN-2

Example

```
#Include ADwinPro_All.inc
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).
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_Con(1,1,8)

  If (Par_9 = 40) Then
    REM New message for message object with the identifier 40
    REM has arrived
    Par_1 = CAN_Msg[1]       'Read out high-byte
    For n = 2 To 4           'Combine with remaining 3 bytes to
     Par_1 = Shift_Left(Par_1,8) + CAN_Msg[n]'a 32-bit value
    Next n
    REM Convert the bit pattern in Par_1 to data type FLOAT and
    REM assign to the variable FPar_1.
    FPar_1 = Cast_LongToFloat(Par_1)
  EndIf
```

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

module	Specified module address (1…255).	LONG
channel	Number (1, 2) of the CAN channel, that determines the CAN controller.	LONG
rate	Baud rate of the CAN controller: High speed CAN: 5000…1000000 Bit/s. Low speed CAN: 5000 … 125000 Bit/s.	FLOAT
ret_val	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 synchroniziation 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 (1MBit/s).

See also

Get_CAN_Reg, Init_CAN, Set_CAN_Reg

Valid for

CAN-1, CAN-2

Example

```
#Include ADwinPro_All.Inc
Dim status As Long
Init:
  Init_CAN(1,1)              'Initialize 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 controll register has the address 0 and the status register the address 1.

See also

Init_CAN, Set_CAN_Baudrate, Get_CAN_Reg

Valid for

CAN-1, CAN-2

Example

```
#Include ADwinPro_All.Inc
Init:
  Init_CAN(1,1)              'Initialize CAN controller
  Set_CAN_Reg(1,1,0,1)       'Set control register to the value 1
```

Transmit

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,msg_no)
```

Parameters

module	Specified module address (1…255).	LONG
channel	Number (1, 2) of the CAN channel, that determines the CAN controller.	LONG
msg_no	Number (1… 14) of the message object in the CAN controller.	LONG

Notes

To send a message, follow these steps:

- Enable the message object for sending with **En_Transmit**.
- Enter the message into the array **CAN_Msg**: data bytes and the number of data bytes.
- Before sending: Query with **Transmit_Status**, if the message object is ready to send.
- Send the message with **Transmit**.

The CAN interface sends the message as soon as the message object has received access rights to the CAN bus.

See also

CAN_Msg, En_Receive, En_Transmit, Read_Msg, Transmit_Status

Valid for

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_Status

Transmit_Status returns if a message object is ready to send.

Syntax

```
#Include ADwinPro_All.Inc
ret_val = Transmit_Status(module,channel,msg_no)
```

Parameters

module	Specified module address (1…255).	LONG
channel	Number (1, 2) of the CAN channel, that determines the CAN controller.	LONG
msg_no	Number (1… 14) of the message object in the CAN controller.	LONG
ret_val	Status of message object. 0: Ready to send. 1: Not ready to send.	LONG

Notes

The return value is only useful for message objects, which are configured to send.

A message object, which is not ready to send still contains a message, which has to be sent or is being sent.

The CAN interface sends the message as soon as the message object has received access rights to the CAN bus.

You can transmit a messages also without querying the status of the message object. But if you generate messages faster than the CAN controller my transmit them, single messages will be lost.

See also

CAN_Msg, En_Receive, En_Transmit, Read_Msg, Transmit

Valid for

CAN-1, CAN-2

Example

```
#Include ADwinPro_All.inc

Init:
  Init_CAN(1,1)              'initialize CAN controller
  En_Transmit(1,1,6,40,0)    'initialize message object 6
  Par_1 = 0
  CAN_Msg[1] = Par_1         'set message value
  CAN_Msg[9] = 1             'message length in bytes

Event:
  Inc(Par_1)
  CAN_Msg[1] = Par_1         'set message value
  If (Transmit_Status(1,1,6) = 0) Then 'ready to send?
    Transmit(1,1,6)          'send message object 6
  EndIf
  If (Par_1 = 255) Then Par_1 = 0
```

2.5.3 Field bus

This section describes instructions, which apply to Pro I Field bus modules:

Changed_Data

Changed_Data checks, if the data 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**).

Valid for

Inter-SL, Profi-DP-SL, Profi-IRT-CU, Profi-IRT-FO

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
Rem The example premises an initialization

Event:
  Request_Access(1,2)       'Request access to the DP-RAM
                            '(Output area)
  Do
    Par_1 = Check_Access(1) 'Get access rights status
  Until (Par_1 <> -1)
  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 (Pro I)" on page 230.

Check_Access

`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

module	Specified module address (1…4).	LONG
ret_val	Status of access right: -1: Access right request not processed yet. ≥0: 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.

The access right can only be requested again, if a previous access request is processed completely.

Valid for

Inter-SL, Profi-DP-SL, Profi-IRT-CU, Profi-IRT-FO

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
Rem The example premises an initialization

Event:
  Request_Access(1,1)       'Request access to the DP-RAM
                            '(Control register).
  Do
    Par_1 = Check_Access(1) 'Get access rights status
  Until (Par_1 <> -1)
  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 (Pro I)" on page 230.

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…255).	LONG

Notes

The function accesses each memory location, regardless wether 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.

Valid for

Inter-SL, Profi-DP-SL, Profi-IRT-CU, Profi-IRT-FO

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

`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.	LONG
		LONG
`start`	Index (0…244), which determines both: First read array element as well as the first written memory byte in DP-RAM.	LONG
`count`	Number of data bytes transferred into DP-RAM.	LONG

Notes

You may only use this instruction when the application has the access right for the DP-RAM.

The destination array must already be declared, at least with `start` + `count` array elements. The first data byte is read from address `start` in DP-RAM, while the first written array element is `array[start+1]`.

The data byte is written into bits 0…7 of the array elements (lowest byte), all other bits are 0.

Valid for

Inter-SL, Profi-DP-SL, Profi-IRT-CU, Profi-IRT-FO

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
Rem The example premises an initialization

Event:
  Request_Access(1,2)       'Request access to the DP-RAM
                            '(output area)
  Do
    Par_1 = Check_Access(1) 'Get access rights status
  Until (Par_1 <> -1)
  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 (Pro I)" on page 230.

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, reserved)
```

Parameters

module — Specified module address (1…4). LONG

IO_in — Size of the input data area for cyclic data. LONG

Inter-SL: 1…10 words (1 word = 2 bytes).
other modules: 0…244 bytes
(IO_in + IO_out: 1…416 bytes).

Par_in — Length of the input data area for acyclic data. LONG

Inter-SL (values in words):
0: Module runs as digital slave.
1…200; standard 32. Module runs as PCP participant.

other modules: 0 (acyclic data is not supported).

IO_out — Length of the output data area in bytes for cyclic data. LONG

Inter-SL: 1…10 words (1 word = 2 bytes).
other modules: 0…244 bytes
(IO_in + IO_out: 1…416 bytes).

Par_out — Length of the output data area for acyclic data. LONG

Inter-SL (values in words):
0: Module runs as digital slave.
1…200; standard 32. Module runs as PCP participant.

other modules: 0 (acyclic data is not supported).

in_hdl — Bit pattern for the handling of the input data area: LONG

Bit no.	31:2	1	0
Bit = 0	–	Disable flag for **Changed_Data**.	Clear input area as soon as the application stops
Bit = 1.	–	Enable flag for **Changed_Data**.	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.

reserved	Always enter 0. This parameter is required for reasons of compatibility	LONG
ret_val	Bit pattern as status of the initialization: Bit = 0: no error. Bit = 1: error occurred (meaning see below).	LONG

Bit no.	31:16	15	14	13:8	7	6	5:3	2	1	0
Error	–	A_7	A_6	–	A_5	A_4	–	A_3	A_2	A_1

A_1: Error during hardware check
A_2: Error during start of the initialization
A_3: Error during the initialization
A_4: Error in the DP-RAM
A_5: Error at the end of initialization
A_6: No message received
A_7: 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; find more information in the hardware manual.

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 initializing in a low-priority process or in a process section with low priority (**LowInit:**).

Valid for

Inter-SL, Profi-DP-SL, Profi-IRT-CU, Profi-IRT-FO

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,
  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 instable.

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.

The access right can only be requested again, if a previous access request is processed completely.

Valid for

Inter-SL, Profi-DP-SL, Profi-IRT-CU, Profi-IRT-FO

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
Rem The example premises an initialization

Event:
  Request_Access(1,1)       'Request access to the DP-RAM
                            '(Control register)
  Do
    Par_1 = Check_Access(1) 'Get access rights status
  Until (Par_1 <> -1)
  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 (Pro I)" on page 230.

Request_Release_Access

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

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 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.

The access right can only be requested again, if a previous access request is processed completely.

Valid for

Inter-SL, Profi-DP-SL, Profi-IRT-CU, Profi-IRT-FO

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
Rem The example premises an initialization

Event:
  Request_Access(1,1)       'Request access to the DP-RAM
                            '(Control register).
  Do
    Par_1 = Check_Access(1) 'Get access rights status
  Until (Par_1 <> -1)
  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 (Pro I)" on page 230.

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 wether 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.

Valid for

Inter-SL, Profi-DP-SL, Profi-IRT-CU, Profi-IRT-FO

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

`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 source array.	LONG
		LONG
`start`	Index (0…244), which determines both: First read array element as well as the first written memory byte in DP-RAM.	LONG
`count`	Number of data bytes transferred into DP-RAM.	LONG

Notes

The instruction can only be used, when the application has the access rights for the DP-RAM.

The source array must already be declared, at least with `start`+`count` array elements. The first data byte is read from `array[start+1]`, while the first written memory address is `start`.

Only bits 0…7 of the array elements are transferred. (lowest byte).

Valid for

Inter-SL, Profi-DP-SL, Profi-IRT-CU, Profi-IRT-FO

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
Rem The example premises an initialization

Event:
  Request_Access(1,4)       'request access to the DP-RAM
                            '(input area)
  Do
    Par_1 = Check_Access(1) 'Get access rights status
  Until (Par_1 <> -1)
  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 (Pro I)" on page 230.

2.5.4 RSxxx

This section describes instructions, which apply to Pro I RSxxx modules:

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,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	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 using 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 availabe in the include file.

See also

Read_FIFO, RS_Init, RS_Reset, RS485_Send, Set_RS, Write_FIFO

Valid for

RS232-2, RS232-4, RS422-2, RS422-4, RS485-2, RS485-4

Example

```
#Include ADwinPro_All.Inc

Event:
  Rem …
  Rem check if channel 1 still has data to send
  Par_1 = Check_Shift_Reg(1,1)
  Rem …
```

Get_RS

`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 using 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 availabe in the include file.

See also

Check_Shift_Reg, Read_FIFO, RS_Init, RS_Reset, RS485_Send, Set_RS, Write_FIFO

Valid for

RS232-2, RS232-4, RS422-2, RS422-4, RS485-2, RS485-4

Example

- / -

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

Valid for

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 (Pro I) from page 232.

RS_Init

`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

`module`	Specified module address (1…4).	LONG
`channel`	Channel, which is to be initialized (1, 2 or 1…4).	LONG
`baud`	Transfer rate in Baud: RS232: 35 … 115200 Baud. RS485: 35…2304000 Baud.	LONG
`parity`	Use of test bits: 0: without parity bit. 1: even parity. 2: odd parity.	LONG
`bits`	Amount of data bits (5, 6, 7 or 8).	LONG
`stop`	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
`handshake`	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 `RS_Reset`.

The baud rates are derived from the basic clock rate of 2304 MHz 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

Common baud rates [Bit/s]	
115200	4800

See also

Check_Shift_Reg, Get_RS, Read_FIFO, RS_Reset, RS485_Send, Set_RS, Write_FIFO

Valid for

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 (Pro I) on page 232.

RS_Reset

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.

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

Valid for

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 (Pro I) from page 232.

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	Tranfer 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

Valid for

RS485-2, RS485-4

Example

See Example "RS485: Receive and send" on page 235.

Set_RS

`Set_RS` writes a value into a specified register on the specified module.

Syntax

```
#Include ADwinPro_All.Inc
Set_RS(module,register,value)
```

Parameters

`module`	Specified module address (1…4).	LONG
`register`	Number of the register, into which data are written.	LONG
`value`	Value to be written into the register.	LONG

Notes

We recommend using 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 availabe in the include file.

See also

Check_Shift_Reg, Get_RS, Read_FIFO, RS_Init, RS_Reset, RS485_Send, Write_FIFO

Valid for

RS232-2, RS232-4, RS422-2, RS422-4, RS485-2, RS485-4

Example

- / -

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, RS422-2, RS422-4, 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 (Pro I) from page 232.

3 Program Examples

The following examples are available:

Most examples are stored as program files in the directory <C:\ADwin\ADbasic\samples_ADwin_Pro> or <C:\ADwin\ADbasic\samples_ADwin_ProII>.

3.1 Online Evaluation of Measurement Data (Pro I)

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 and 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
#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
  Processdelay = 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      '1000 samples done?
    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
    Par_10 = 1             'set End-Flag
  EndIf
```

3.2 Digital Proportional Controller

The program <PRO_DMO2.BAS> is a digital proportional controller. The setpoint 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
#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
  Processdelay = 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
```

3.3 Data Exchange with DATA arrays (Pro I)

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
Dim Data_1[1000] As Integer
Dim index As Integer

Init:
 Par_10 = 0
 index = 0                  'reset array pointer
 Processdelay = 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
```

3.4 Digital PID Controller (Pro I)

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`	differentation time of the controller
`Par_1`	Setpoint in digits
`Par_6`	controller sampling rate in units of 25 ns

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
#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
  Processdelay = 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
```

3.5 Data exchange with fieldbus (Pro I)

The following program is an example for exchanging data with the fieldbus.

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
#Define num_data 10          'number input and output data
#Define module 1             'module address
Dim Data_1[num_data] As Long 'array for output data
Dim Data_2[num_data] As Long 'array for input data
Dim temparr[num_data] As Long 'array for temporary data
Dim i, seq_step As Long
Dim range, ret_val As Long

Init:
  Rem clock rate of this process must be 10Hz at least
  Processdelay = 100000      '400 Hz for Processor T9
  mem_area = 6               'Memory area to be accessed
                             '(input/output data)
  seq_step = 0               'current step: start anew
  Rem Initialize; with Profinet use Init_Slave_Profinet
  ret_val = Init_Slave(module, num_data, 0, num_data, 0, 2, 2, 0)
  Rem ret_val = Init_Slave_Profinet(module, num_data, 0,
    num_data, 0, 2, 2, 0)
  If (ret_val <> 0) Then Exit 'Initialize error
  For i = 1 To num_data      'Initialize sending data
   Data_1[i] = i
  Next i

Event:
  SelectCase seq_step
  Case 0                     'start anew
   Request_Access(module,mem_area)'request access to memory area
   seq_step = 1
  Case 1                     'Check access right
   ret_val = Check_Access(module)
   If(ret_val <> -1)Then     'request already processed?
    If (ret_val = mem_area) Then 'if access right is given...
     seq_step = 2            '...start data exchange
    Else
     seq_step = 0            '...else skip and start anew
    EndIf
   EndIf
  Case 2                     'data exchange with Fieldbus
   Set_Write_Buffer(module, Data_1, 1, num_data) 'send data
   Get_Read_Buffer(module, temparr, 200h, num_data) 'read data
   ret_val = Check_Access(module)
   Rem If access right is given, read data is valid
   If (ret_val = mem_area) Then
    For i = 1 To num_data    'copy read data
     Data_2[i] = temparr[i]
    Next i
    Request_Release_Access(module,mem_area) 'release access
    seq_step = 3
   Else 'else skip and start anew, skip data
    seq_step = 0
   EndIf
  Case 3                     'access right released?
   ret_val = Check_Access(module)
   If(ret_val <> -1)Then     'request already processed??
    If (ret_val = 0) Then    'if access right is released...
     seq_step = 0            '...start anew
    Else                     '...else release access again
     Request_Release_Access(module, mem_area)
    EndIf
   EndIf
  EndSelect
```

3.6 Examples for RS232 and RS485 (Pro I)

The following examples are complete programs for sending and receivingof data and strings with RS232 or RS485.

RS232:
Send and receive

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
```

RS232:
Send string instruction

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** section, 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_All.Inc

REM import string library
#If Processor = 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:
  Processdelay = 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_All.Inc

REM import string library
#If Processor = 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:
  Processdelay = 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 RS485module 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
```

Instruction Lists

A.1 Alphabetic Instruction List

R

S

T

W

A.2 Instruction List sorted by Module Types

You find the instruction lists of the modules on these pages:

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-4/16 Rev. B

AIn-F-8/12 Rev. A

AIn-F-8/14 Rev. B

AIn-F-8/16 Rev. A

AIn-F-8/16 Rev. B

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-VR2PW2

CNT-VR4L(-I)

CNT-VR4(-I)

CO4-D

CO4-I

CO4-T

Comp-16 Rev. A

CPU-T10

CPU-T9

DIO-32 Rev. A

DIO-32 Rev. B

Inter-SL

LS-2 Rev. A

OPT-16 Rev. A

OPT-16 Rev. B

Profi-DP-SL

Profi-IRT-CU

Profi-IRT-FO

PT100-4, PT100-8

PWM-4(-I)

REL-16 Rev. A

REL-16 Rev. B

RS232-2, RS232-4

RS422-2, RS422-4

RS485-2, RS485-4

Storage Rev. A

TC-16

TC-4

TC-8

TC-8-ISO Rev. E

TRA-16 Rev. A

TRA-16 Rev. B

(LP)SH-8(-FI)

A.3 Thematic Instruction List

Die Befehle sind in die folgenden Themengruppen aufgeteilt. Innerhalb der Themengruppen sind die Befehle alphabetisch sortiert.

Analog Inputs

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.
ADCF	executes a complete measurement on a Fast-ADC.
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.
Burst_CStart	starts a burst-measurement sequence in the "Continuous" mode on the specified module.
Burst_Init	sets the parameters for a burst-measurement sequence on the specified module.
Burst_Read	copies the measurement values of a channel into a specified array.
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.
Burst_Start	starts a burst-measurement sequence on the specified module (independent of the processor).
Burst_Status	determines the amount of the burst-measurements, which are still to be executed on the specified module.
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.
Read_ADCF4	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.
Read_ADCF8	reads out the conversion results from all 8 F-ADC of the specified module.
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.
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).
Seq_Read	copies a specified amount of measurement values (16-bit each) from the module to a destination array.

Seq_Read32	copies all 32 measurement values (16-bit each) from the specified module into the destination array.
Seq_Read_One	reads out a specified measurement value (16 bit) of a channel group on the specified module.
Seq_Read_Packed	copies an even amount of measurement values (16-bit each) in pairs from the specified module to a destination array.
Seq_Read_Two	reads out at the same time 2 consecutive measurement values (16-bit each) of a channel group, on the specified module and returns them in a 32-bit value.
Seq_Select	determines the channels belonging to the channel group, which will be converted by the sequential control on the specified module.
Seq_Set_Delay	determines the settling time of the sequential control on the specified module.
Seq_Status	determines how many channels of the channel group are already converted and stored by the sequential control of the specified module.
Set_Gain	sets the operating mode for a channel of the specified module, and thus the gain factor and measurement range, too.
Set_Mux	sets the multiplexer input of the module to a specified channel and gain.
SE_Diff	sets the operating mode single ended or differential for all analog inputs on the specified module.
SH_SetMode	sets the mode of the sample and hold levels.
Start_Conv	starts the A/D conversion on the specified module.
Start_ConvF	starts the conversion of one or more F-ADCs of the specified module.
Sync_Mode	determines on the specified module the type of synchronization with other modules, especially for burst-measurement sequences.
Wait_EOC	waits, until the last A/D conversion has finished.
Wait_EOCF	waits until the end of conversion on all specified F-ADCs.

Analog Outputs

DAC	outputs an (analog) voltage on the channel of the specified module, that corresponds to the indicated digital value.
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.
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 Start_DAC.
WriteDAC32	writes 2 digital values into 2 output registers of a DAC on the specified module. The conversion into output voltage is started by Start_DAC.

CAN-Bus

CAN_Msg	is a one-dimensional array consisting of 9 elements, where the message objects of the CAN bus are saved during sending and receiving.
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.
Transmit_Status	returns if a message object is ready to send.

Communication interface

CAN bus

Read_Msg_Con	returns if a completely new message has been received in a message object of one of the CAN controllers on the module.

Fieldbus

Changed_Data	checks, if the data 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.

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. 40 kHz 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 speicified 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	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	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 given 32-bit value, 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	Dig_ReadLatch 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	GET_Digout_LONG returns the contents of the output-latch (register for digital outputs) on the specified module.
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.
Get_Digout_Word2	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	PWM_OUT sets a specified PWM output channel on the specified module to the level "high" or "low".
PWM_Set	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.
CPU_Digin	Processor T9 and T10 only. CPU_Digin returns, wether a falling edge arose at the input Digin 0 of the processor module since the last call of the instruction.
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.
ResetWatchdogTimer	resets 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.
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 bbeen 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.

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	calculates the temperature in degrees Celsius or Fahrenheit from the measured digital value of a thermo couple type J.
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	returns the thermoelectric voltage (µV) or the temperature (°C / °F) of a specified channel on the module.
TC_Read_E	returns the thermoelectric voltage (µV) or the temperature (°C / °F) of a specified channel on the module.
TC_Read_J	returns the thermoelectric voltage (µV) or the temperature (°C / °F) of a specified channel on the module.

TC_Read_K	returns the thermoelectric voltage (µV) or the temperature (°C / °F) of a specified channel on the module.
TC_Read_N	returns the thermoelectric voltage (µV) or the temperature (°C / °F) of a specified channel on the module.
TC_Read_R	returns the thermoelectric voltage (µV) or the temperature (°C / °F) of a specified channel on the module.
TC_Read_S	returns the thermoelectric voltage (µV) or the temperature (°C / °F) of a specified channel on the module.
TC_Read_T	returns the thermoelectric voltage (µV) or the temperature (°C / °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.