ADwin-Pro

System specifications
Programming in *ADbasic*

ADwin-Pro Software, manual version 2.4, December 2004

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

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

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

Table of contents

Typographical Conventions

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

You find a "note" next to

– information, which have absolutely to be considered in order to guarantee an operation without any errors
– advice for efficient operation

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

<C:\ADwin\ …>

File names and paths are placed in parentheses and characterized in the font Courier New.

Program text

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

Var_1

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

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

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

1 Introduction

With the real-time development tool *ADbasic* you have purchased 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 multiprocessing capacities of the system.

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\ADbasic3\inc>`.

In order to get access to the *ADwin-Pro* modules you have to include - according to the functions specified - one or more of the following files into your *ADbasic* program.

`ADWINPRO.INC`	System instructions
`ADWPAD.INC`	Instructions for access to the A/D-converter modules
`ADWPDA.INC`	Instructions for access to the D/A-converter modules
`ADWPDIO.INC`	Instructions for access to the digital modules
`ADWPEXT.INC`	Instructions for access to the add-on modules

With the *ADbasic* instruction **`#INCLUDE`** the files will be included.

Please note:

To have your *ADwin* systems work properly, keep strictly to the information given 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 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 instructions

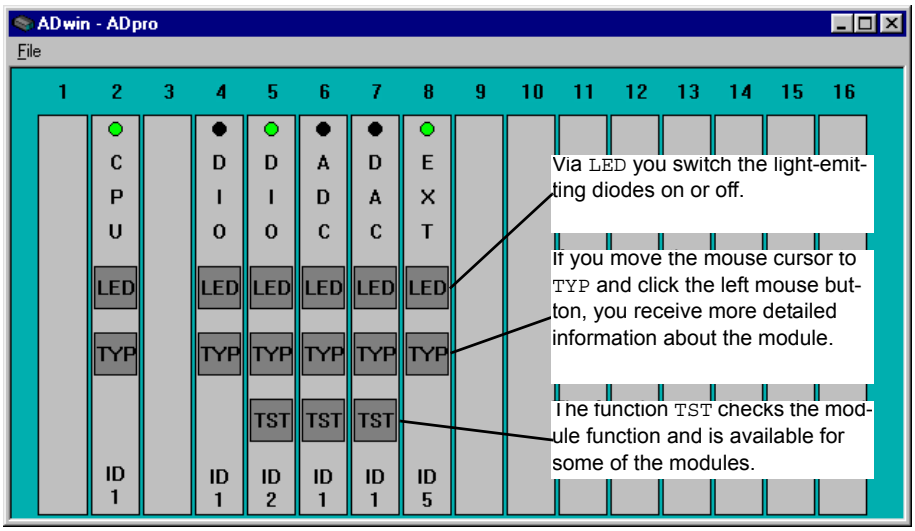
OEM products are mentioned without referring to possible patent rights, whose existence is not to be excluded.

Subject to change.

Hotline address: see inner side of cover page

2 The Test Program ADpro

The test and diagnostic program <ADpro.exe> shows the modules your *ADwin-Pro* system is equipped with. Each module identifies the module address (ID). Moreover, the function of the modules is checked.



The following table shows the module classes and the information for the module classes that are indicated by the test function (TST).

Module class	Description	Testfunction (TST)
CPU	Processor module	Not available
ADC	Analog-to-digital converter module	Displays the current measurement values of max. 8 inputs (in digits).
DAC	Digital-to-analog converter module	For every output an output value can be defined (in digits)
DIO	Digital input/output module or counter module	Displays the current signal status at the inputs (decimal or binary), the digital outputs can be set or cleared.
EXT	Add-on module such as thermocouples, filters ...	Not available

Depending on the processor the program <ADpro.exe> uses one of the following files:

- ADprot.t90 Processor T9 (ADSP21062)
- ADprot.ta0 Processor T10 (ADSP21162)

Each of the files contains the necessary *ADbasic* process.

3 *ADbasic* instruction for *ADwin-Pro* modules

The instructions for the access to the *ADwin-Pro* modules are to be found in the include files. The instructions are sorted according to the include files and then alphabetically.

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

– Alphabetic Instruction List
– Thematic Instruction List
– Instruction List sorted by Module Types

For each instruction the description includes syntax, parameter, hints to related instructions, a module list (the instruction can be used for) and an example.

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

3.1 ADWINPRO.INC

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

```
#INCLUDE ADwinpro.inc
```

in your *ADbasic* process, you will be able to apply all functions and procedures of this file. On the following pages all `<ADwinpro.inc>` functions will be described more detailed.

The instructions need the parameter `mod_class` that determines the module class. Use the following constants for this parameter, which are defined in the file `<ADwinpro.inc>`:

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

CHECKLED

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

Syntax

```
#INCLUDE ADWINPRO.INC
ret_val = CHECKLED(module,mod_class)
```

Parameters

module	Specified module address	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 whereof the status cannot be returned with this instruction.

See also

SETLED

To be used for the modules

all modules (except MB8)

Example

```
#INCLUDE ADWINPRO.INC

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

CPU_DIGIN

The instruction **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.INC
ret_val = CPU_DIGIN()
```

Parameters

ret_val	Flag, if a rising edge has been detected at the input Digin 0:	LONG
	0: No rising edge detected	
	1: Rising edge has been detected at least once	

Notes

The instruction **CPU_DIGIN** reads the module's internal flag for falling edges; doing so, the flag will be automatically reset to the value 0.

To be used for the modules

CPU-T9 (ordering option), CPU-T10

Example

```
#INCLUDE ADWINPRO.INC
DIM dummy AS LONG

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

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

EVENTENABLE

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.

Syntax

```
#INCLUDE ADWINPRO.INC
EVENTENABLE(module,mod_class,value)
```

Parameters

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

Notes

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

Most of the modules have an event input. As soon as you have enabled the event input with **EVENTENABLE**, a rising edge at this input is recognized as event signal and the specified process responds.

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

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

To be used for the modules

all modules with an event input (module group DIO)

Example

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

EVENT:
  ...
```

RESETWATCH DOGTIMER

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

Syntax

```
#INCLUDE ADWINPRO.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.

The time needed for counting from the start value to 0 depends on the processor type:

Processor type	Duration
T4 and T8	80 ms
ADSP (T9, T10)	1.600 ms

Set the active watchdog timer at least once to the start value within this time 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

STARTWATCH DOG, STOPWATCHDOG

To be used for the modules the modules

CPU-T9, CPU-T10

Example

```
#INCLUDE ADWINPRO.INC

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

SETLED

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

Syntax

```
#INCLUDE ADWINPRO.INC
SETLED(module,mod_class,value)
```

Parameters

module	Specified module address	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

To be used for the modules

all modules (except MB8)

Example

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

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

STARTWATCH DOG

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

Syntax

```
#INCLUDE ADWINPRO.INC
STARTWATCHDOG()
```

Notes

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

The time needed for counting from the start value to 0 depends on the processor type:

Processor type	Duration
T4 and T8	80 ms
ADSP (T9, T10)	1.600 ms

Set the active watchdog timer at least once to the start value within this time interval, in order to keep your Pro system working (see **RESETWATCHDOGTIMER**).

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

See also

RESETWATCH DOGTIMER, STOPWATCHDOG

To be used for the modules the modules

CPU-T9, CPU-T10

Example

```
#INCLUDE ADWINPRO.INC
INIT:
  STARTWATCHDOG()           'Activate watchdog
```

STOPWATCHDOG

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

Syntax

```
#INCLUDE ADWINPRO.INC
STOPWATCHDOG()
```

Notes

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

See also

RESETWATCH DOGTIMER, STARTWATCH DOG

To be used for the modules the modules

CPU-T9, CPU-T10

Example

```
#INCLUDE ADWINPRO.INC
INIT:
  STARTWATCHDOG()           'Activate watchdog
  …

EVENT:
  ResetWATCHDOG()           'Reset watchdog
  …

FINISH:
  STOPWATCHDOG()            'Disable watchdog
```

SYNCALL

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

Syntax

```
#INCLUDE ADWINPRO.INC
SYNCALL()
```

Notes

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

Module type	Action
Analog input	Start A/D conversion on all ADCs (for **SYNCENABLE** =1) or Start burst measurement sequence (for **SYNCENABLE** =3 and only for the modules Pro-AIn-F-x/14)
Analog output	Start D/A conversion with the value from the DAC register (see **WRITEDAC**)
Digital input	Transfer current status of the inputs into the input temporary register (read out the values there with **DIG_READLATCH1**, **DIG_READLATCH2**).
Digital output	Read the value from the output temporary register and transfer it to the digital output (see **DIG_WRITELATCH1**, **DIG_WRITELATCH2**).
Counter	Transfer current counter values finto the counter temporary register (read out the values there with **CNT_READLATCH16**, **CNT_READLATCH32**, **CO4_READLATCH**).

See also

SYNCENABLE, SYNCSTAT

To be used for the modules

AIn-x/x, AOut-x/x, CNT-x/x

PWM-4, CO4, DIO32, OPT-16, TRA-16, REL-16, (LP)SH-8

Example

```
#INCLUDE ADWINPRO.INC
#INCLUDE ADWPAD.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

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

Syntax

```
#INCLUDE ADWINPRO.INC
SYNCENABLE(module,mod_class,value)
```

Parameters

module	Specified module address	LONG
mod_class	Module class: One of the constants ad, da, dio, cpu or ext	LONG
value	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

To be used for the modules

AIn-x/x, AOut-x/x, CNT-x/x

PWM-4, CO4, DIO32, OPT-16, TRA-16, REL-16, (LP)SH-8

Example

```
#INCLUDE ADWINPRO.INC
#INCLUDE ADWPDIO.INC
#INCLUDE ADWPAD.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 the 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

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

Syntax

```
#INCLUDE ADWINPRO.INC
ret_val = SYNCSTAT(module,mod_class)
```

Parameters

module	Specified module address	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

To be used for the modules

AIn-x/x, AOut-x/x, CNT-x/x

PWM-4, CO4, DIO32, OPT-16, TRA-16, REL-16, (LP)SH-8

Example

```
#INCLUDE ADWINPRO.INC
#INCLUDE ADWPDA.INC
DIM i AS LONG
DIM DATA_1[1000], DATA_2[1000] AS LONG

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

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

3.2 ADWPAD.INC

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

```
#INCLUDE ADWPAD.INC
```

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

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

In the list of modules you will find which of the functions corresponds to the *ADwin-Pro* modules.

ADC

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

Syntax

```
ret_val = ADC(module,input_no)
#INCLUDE ADWPAD.INC
```

Parameters

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

Notes

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

SET_MUX	→	…	→	START_CONV	→	WAIT_EOC	→	READADC
Set the multiplexer to the specified input		Wait for settling of the multiplexer (3µs) 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: **GLOBALDELAY** < 200 (see above).
- High internal resistance (>3kΩ) of the voltage source of the measurement signal: This increases the settling time of the multiplexer to more than 3.0µs or 14µs.
- You would like to use waiting times for additional program tasks.

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

A process with low priority can be interrupted in the middle of an **ADC** sequence by a process with higher priority. When the process with higher priority sets the multiplexer to another channel during this interruption, the function of the process with low priority returns the measurement value from the wrong channel.
Several parallel processes with high priority can apply the function **ADC** without any problems, because processes with high priority do not interrupt each other.

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

See also

SET_MUX, START_CONV, WAIT_EOC, READADC, ADC16

To be used for the modules

Pro-AIn-8/12 Rev. A/B, Pro-AIn-8/14 Rev. A

Pro-AIn-8/16 Rev. B/C

Pro-AIn-32/12 Rev. A/B, Pro-AIn-32/14 Rev. A

Pro-AIn-32/16 Rev. B/C

Pro-AO-16/8-12 Rev. A

Pro-LPSH-4-FI, Pro-LPSH-8-FI, Pro-LPSH-8

Example

```
#INCLUDE ADWPAD.INC
DIM value AS LONG
EVENT:
  value = ADC(1, 4)          'Measure a value at the analog input 4
```

ADC16

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

Syntax

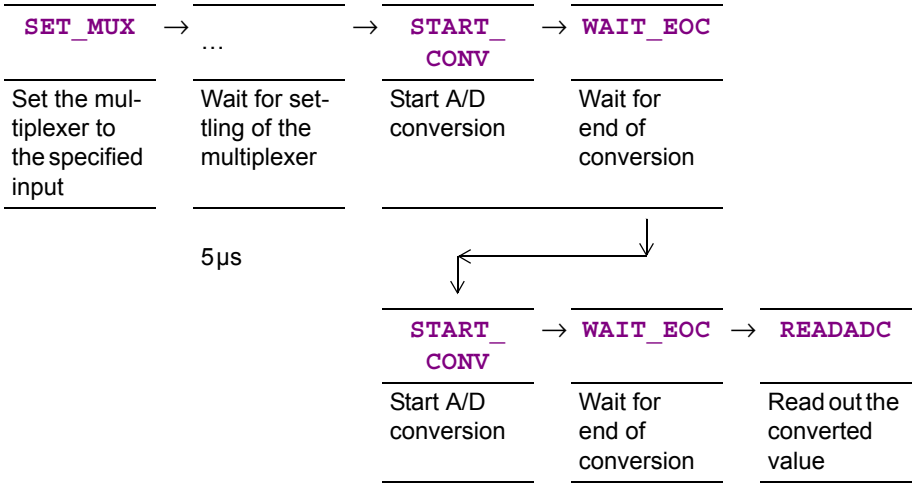
```
#INCLUDE ADWPAD.INC
ret_val = ADC16(module,input_no)
```

Parameters

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

Notes

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



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

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

Start conversion twice

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

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

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

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

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

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

See also

SET_MUX, START_CONV, WAIT_EOC, READADC, ADC

To be used for the modules

Pro-AIn-8/16

Pro-LPSH-4-FI, Pro-LPSH-8-FI, Pro-LPSH-8

The instruction **ADC16** does not apply for the modules Pro-AIn-8/16 Rev. B or later. Use the instruction **ADC** instead.

Example

```
#INCLUDE ADWPAD.INC
DIM value AS LONG
EVENT:
  value = ADC16(1, 7)       'Measure a value at the analog input 7
```

ADCF

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

Syntax

```
#INCLUDE ADWPAD.INC
ret_val = ADCF(module,input_no)
```

Parameters

module	Specified module address	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

SET_MUX, START_CONVF, WAIT_EOCF, READADCF

To be used for the modules

Pro-AIn-F-4/12 Rev. A, Pro-AIn-F-8/12 Rev. A

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Pro-AIn-F-4/16 Rev. A, Pro-AIn-F-8/16 Rev. A

Example

```
#INCLUDE ADWPAD.INC
DIM value AS LONG

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

BURST_ABORT

The instruction **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 ADWPAD.INC
ret_val = BURST_ABORT(module)
```

Parameters

module	Specified module address	LONG
ret_val	Number of the stored measurement values (0…1048575)	LONG

Notes

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

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

See also

BURST_INIT, BURST_START, BURST_STATUS

To be used for the modules

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Example

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

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

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

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

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

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

BURST_CREAD

The instruction **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 ADWPAD.INC
BURST_CREAD(module,channel,length,array[],arr_idx)
```

Parameters

module	Specified module address	LONG
channel	Channel to be transferred (1…4; with a Pro-AIn-F-8/14 Rev. A: 1…8)	LONG
length	Number of the measurement values to be transferred (1…n), divided by 2	LONG
array[]	Destination array into which the measurement values are transferred	ARRAY 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 + length elements, in order to receive all measurement values.

Read out measurement values of a continuous burst-measurement sequence only with the instruction **BURST_CREAD** (see **BURST_CSTART**).

See also

BURST_INIT, BURST_CSTART, BURST_STATUS, SET_GAIN, SYNC_MODE

Can be also used for the modules

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Example

```
#INCLUDE ADWPAD.INC
#DEFINE length 10000
#DEFINE module 1
#DEFINE sampleperiod 20    '2,000 kHz measurement rate
                           '[=1/(25ns*20)]
DIM DATA_1[length] AS LONG

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

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

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

BURST_CSTART

The instruction **BURST_CSTART** starts a burst-measurement sequence in the "Continuous" mode on the specified module.

Syntax

```
#INCLUDE ADWPAD.INC
BURST_CSTART(module)
```

Parameters

module	Specified module address	LONG

Notes

The instruction uses the built-in memory as ring buffer. During a continous 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

To be used for the modules

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Example

```
#INCLUDE ADWPAD.INC
#DEFINE length 10000
#DEFINE module 1
#DEFINE sampleperiod 20    '2,000 kHz measurement rate
                           '[=1/(25ns*20)]
DIM DATA_1[length] AS LONG

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

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

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

BURST_INIT

The instruction **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 ADWPAD.INC
BURST_INIT(module,mode,pulses,samples)
```

Parameters

`module` — Specified module address — LONG

`mode` — Measurement mode (1…3; with a Pro-AIn-F-8/14 Rev. A: 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 = 1048575 = 0FFFFFH
2	1, 2	2^19 - 1 = 524287 = 7FFFFH
3	1…4	2^18 - 1 = 262143 = 3FFFFH
4	1…8	2^18 - 1 = 131071 = 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

To be used for the modules

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

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 ADWPAD.INC
#INCLUDE ADWINPRO.INC
#DEFINE samples 10000       'Amount of measurements to be executed
#DEFINE ainadr 1            'Address of AIn module
#DEFINE sampleperiod 40     '1.0 MHz measurement rate
                            '=1/(25ns*40)
DIM DATA_1[samples] AS LONG
DIM DATA_2[samples] AS LONG
DIM dig_value AS LONG       'Supplied voltage value
DIM remaining AS LONG       'Amount of the remaining measurement
                            'values
DIM run_state AS LONG       'Status of the measurement sequence:
                            'not running/running/finished

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

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

The instruction **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 ADWPAD.INC
BURST_READ(module, channel, start, length,
           array[], array_idx)
```

Parameters

module	Specified module address	LONG
channel	Channel to be transferred (1…4; with a Pro-AIn-F-8/14 Rev. A: 1…8)	LONG
start	Source startindex: Elements from which it is started to read out the measurement values	LONG
length	Amount of the measurement values to be transferred (1…n), divided by 2	LONG
array[]	Destination array, into which the measurement values are transferred	ARRAY 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 + length-1 elements, in order to receive all measurement values.

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

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

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

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

See also

BURST_ABORT, BURST_INIT, BURST_READ_PACKED, BURST_START, BURST_STATUS, SET_GAIN, SYNC_MODE

To be used for the modules

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

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 ADWPAD.INC
#INCLUDE ADWPDIO.INC
#DEFINE samples 10000       'Number of measurements to be executed
#DEFINE sampleperiod 30     '1.33 MHz meas. rate [=1/(25ns*30)]
#DEFINE dioadr 1            'Module address DIO module
#DEFINE ainadr 1            'Module address AIN module
#DEFINE run_state PAR_1     'Status of the measurement sequence
                            'not running/running/finished
DIM DATA_1[samples] AS LONG
DIM remaining AS LONG       'Number of the remaining measurement
                            'values
DIM i AS LONG
DIM trigger AS LONG         'Marker for external trigger signal

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

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

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

BURST_READ_PACKED

The instruction **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 ADWPAD.INC
BURST_READ_PACKED(module,channel,start,length,
                  array[],array_idx)
```

Parameters

module	Specified module address	LONG
channel	Channel to be transferred (1…4; with a Pro-AIn-F-8/14 Rev. A: 1…8)	LONG
start	Source startindex: Element where you start reading out the measurement values	LONG
length	Amount of the measurement values to be transferred (1…n), divided by 2	LONG
array[]	Destination array, where the measurement values are transferred	ARRAY LONG
array_idx	Destination startindex: Array element from which you start storing the measurement values (1…n)	LONG

Notes

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

Address in the destination 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 + length	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.

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

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

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

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

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

See also

BURST_ABORT, BURST_INIT, BURST_READ, BURST_START, BURST_STATUS, SET_GAIN, SYNC_MODE

To be used for the modules

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Example

```
REM Attention: Measurement in high-priority process, read out in
REM the low-priority section!
#INCLUDE ADWPAD.INC
#INCLUDE ADWINPRO.INC
#DEFINE samples 10000      'Amount of measurements to be executed
#DEFINE ainadr 2           'Address of the AIN module
#DEFINE sampleperiod 40    '1 MHz measurement rate
                           '=1/(25ns*40)
DIM DATA_1[samples] AS LONG
DIM DATA_2[samples] AS LONG
DIM remaining AS LONG      'Amount of the remaining measurement
                           'values
DIM run_state AS LONG      'Status of the measurement sequence:
                           'not running/running/finished
INIT:
  BURST_INIT(ainadr,2,sampleperiod,samples)
                           ''Address, mode,meas. rate,amount of
                           'measurements
  run_state=0              'Measurement sequence is not running
  SET_GAIN(ainadr,1,1)     'Measurement range channel 1 +-5V
  SET_GAIN(ainadr,2,2)     'Measurement range channel 2 +-2.5 V

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

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

BURST_START

The instruction **BURST_START** starts (independent of the processor) a burst-measurement sequence on the specified module.

Syntax

```
#INCLUDE ADWPAD.INC
BURST_START(module)
```

Parameters

module	Specified module address	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 **SYNCALL** synchronously with other measurements.

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

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

   The module must be released with **SYNC_MODE** for synchronization (modes 2 and 3) and with **BURST_START** for starting. The conversions of the measurement sequence are triggered by synchronization signals.

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

   With event-controlled modules (see **SYNC_MODE**), do first release all burst-measurement sequences with **BURST_START** for starting and only then release the event inputs with **EVENTENABLE**.

See also

BURST_ABORT, BURST_INIT, BURST_READ, BURST_READ_PACKED, BURST_STATUS, SET_GAIN, SYNCENABLE, SYNC_MODE

To be used for the modules

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

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 ADWPAD.INC
#INCLUDE ADWINPRO.INC
#DEFINE samples 10000       'Amount of measurement being executed
#DEFINE ainadr 1            'Address of the AIn moduld
#DEFINE sampleperiod 40     '1 MHz measurement rate
                            '=1/(25ns*40)
DIM DATA_1[samples] AS LONG
DIM DATA_2[samples] AS LONG
DIM dig_value AS LONG       'Voltage value
DIM remaining AS LONG       'Amount of the remaining meas. values
DIM run_state AS LONG       'Status of the measurement sequence:
                            'not running/running/finished

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

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

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

BURST_STATUS

The instruction **BURST_STATUS** determines the amount of the burst-measurements which are still to be executed on the specified module.

Syntax

```
#INCLUDE ADWPAD.INC
ret_val = BURST_STATUS(module)
```

Parameters

module	Specified module address	LONG
ret_val	Amount of measurements still to be executed	LONG

Notes

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

See also

BURST_ABORT, BURST_INIT, BURST_CSTART, BURST_CREAD, BURST_READ, BURST_READ_PACKED, BURST_START, SET_GAIN, SYNC_MODE

To be used for the modules

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

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 ADWPAD.INC
#INCLUDE ADWINPRO.INC
#DEFINE samples 10000'Amount of the measurements being executed
#DEFINE ainadr 1'Address of the AIn module
#DEFINE sampleperiod 40'1 MHz measurement rate [=1/(25ns*40)]

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

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

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

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

READADC

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

Syntax

```
#INCLUDE ADWPAD.INC
ret_val = READADC(module)
```

Parameters

module	Specified module address	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

SET_MUX, START_CONV, WAIT_EOC, READADC, ADC16

To be used for the modules

Pro-AIn-8/12 Rev. A/B, Pro-AIn-32/12 Rev. A/B

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

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

Pro-AO-16/8-12 Rev. A

Pro-LPSH-4-FI, Pro-LPSH-8-FI, Pro-LPSH-8

Example

```
#INCLUDE ADWPAD.INC
DIM val1 AS LONG            'Declaration

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

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

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

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

READADC_SCONV

The instruction **READADC_SCONV** reads out the conversion result from an ADC of the specified module and starts immediately a new conversion.

Syntax

```
#INCLUDE ADWPAD.INC
ret_val = READADC_SCONV(module)
```

Parameters

module	Specified module address	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 **SYNCALL** you can no longer use **READADC_SCONV**! Instead, use only the instruction **READADC**, because the conversion is started by **SYNCALL**.

See also

SE_DIFF, SET_MUX, START_CONV, WAIT_EOC, READADC, ADC, ADC16

To be used for the modules

Pro-AIn-8/12 Rev. B, Pro-AIn-8/14 Rev. A

Pro-AIn-32/12 Rev. B, Pro-AIn-32/14 Rev. A

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

Pro-AO-16/8-12 Rev. A

Pro-LPSH-4-FI, Pro-LPSH-8-FI, Pro-LPSH-8

Example

```
#INCLUDE ADWPAD.INC
DIM i AS LONG
DIM DATA_1[1000] AS LONG   'Declaration

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

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

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

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

READADCF

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

Syntax

```
#INCLUDE ADWPAD.INC
ret_val = READADCF(module,adc_no)
```

Parameters

module	Specified module address	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.

See also

ADCF, START_CONVF, WAIT_EOCF, READADCF_32, READADCF_SCONV, READADCF_SCONV_32

To be used for the modules

Pro-AIn-F-4/12 Rev. A, Pro-AIn-F-8/12 Rev. A

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Pro-AIn-F-4/16 Rev. A, Pro-AIn-F-8/16 Rev. A

Example

```
#INCLUDE ADWPAD.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
```

READADCF_32

The instruction **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 ADWPAD.INC
ret_val = READADCF_32(module,adc_no)
```

Parameters

module	Specified module address	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, READADCF_SCONV, READADCF_SCONV_32, START_CONVF, WAIT_EOCF

To be used for the modules

Pro-AIn-F-4/12 Rev. A, Pro-AIn-F-8/12 Rev. A

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Pro-AIn-F-4/16 Rev. A, Pro-AIn-F-8/16 Rev. A

Example

```
#INCLUDE ADWPAD.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

The instruction **READADCF_SCONV** reads out the conversion result from an F-ADC of the specified module and starts immediately a new conversion.

Syntax

```
#INCLUDE ADWPAD.INC
ret_val = READADCF_SCONV(module,adc_nr)
```

Parameters

module	Specified module address	LONG
adc_nr	Number of the ADC to be read (1…4 or 1…8)	LONG
ret_val	Measurement value in the F-ADC register (0…65535).	LONG

Example

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

See also

ADCF, READADCF, READADCF_32, READADCF_SCONV_32, START_CONVF, WAIT_EOCF

To be used for the modules

Pro-AIn-F-4/12 Rev. A, Pro-AIn-F-8/12 Rev. A

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Pro-AIn-F-4/16 Rev. A, Pro-AIn-F-8/16 Rev. A

Example

```
#INCLUDE ADWPAD.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

The instruction **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 ADWPAD.INC
ret_val = READADCF_SCONV_32(module,adc_no)
```

Parameters

module	Specified module address	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,READADCF_32, READADCF_SCONV, START_CONVF, WAIT_EOCF

To be used for the modules

Pro-AIn-F-4/12 Rev. A, Pro-AIn-F-8/12 Rev. A

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Pro-AIn-F-4/16 Rev. A, Pro-AIn-F-8/16 Rev. A

Example

```
#INCLUDE ADWPAD.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

The instruction **SE_DIFF** sets the operating mode single ended or differential for all analog inputs on the specified module.

Syntax

```
#INCLUDE ADWPAD.INC
SE_DIFF(module,choice)
```

Parameters

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

Notes

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

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

See also

ADCF, READADCF, READADCF_32, READADCF_SCONV, READADCF_SCONV_32, START_CONVF, WAIT_EOCF

To be used for the modules

Pro-AIn-32/12 Rev. A/B, Pro-AIn-32/14 Rev. A

Pro-AIn-32/16 Rev. B/C

Example

```
#INCLUDE ADWPAD.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

The instruction **SET_GAIN** sets the operating mode for a channel of the specified module, and thus the gain factor and measurement range, too.

Syntax

```
#INCLUDE ADWPAD.INC
SET_GAIN(module,channel,mode)
```

Parameters

module	Specified module address	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	Measurement 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

To be used for the modules

Pro-AIn-F-4/14 Rev. A

Pro-AIn-F-8/14 Rev. A

Example

```
#INCLUDE ADWPAD.INC
#DEFINE ainadr 1'Module address AIN module

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

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

SET_MUX

The instruction **SET_MUX** sets the multiplexer input of the module to a specified channel and gain.

Syntax

```
#INCLUDE ADWPAD.INC
SET_MUX(module,pattern)
```

Parameters

module	Specified module address	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.

To be used for the modules

Pro-AIn-8/12 Rev. A/B, Pro-AIn-8/14 Rev. A

Pro-AIn-8/16 Rev. A/B/C

Pro-AIn-32/12 Rev. A/B, Pro-AIn-32/14 Rev. A

Pro-AIn-32/16 Rev. B/C

Pro-AO-16/8-12 Rev. A

Pro-LPSH-4-FI, Pro-LPSH-8-FI, Pro-LPSH-8

Example

```
#INCLUDE ADWPAD.INC
DIM val1 AS LONG            'Declaration

EVENT:
  SET_MUX(1,0100010b)       'Set MUX to input 3, set gain 2
                            'Wait 3µs (12-bit ADC) or
                            '14µs (AIn-8/16 Rev. B, AIn-32/16 Rev.
B)
                            'for the settling of the multiplexer
  START_CONV(1)             'Start AD conversion
  WAIT_EOC(1)               'Wait for end of conversion
  val1 = READADC(1)         'Read value from the ADC
```

SEQ_MODE

The instruction **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 ADWPAD.INC
SEQ_MODE(module, mode, gain)
```

Parameters

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

Notes

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

The differences between the modes are as follows:

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

Up to 32 measurement values can be stored in the built-in module memory.
14-bit converters return the measurement result left-aligned in one 16-bit value, that is, the 2 least-significant bits are always zero.

If the internal resistance of the voltage source of the measurement signal is too high (>3kΩ), the predefined settling time of the multiplexer will not be sufficient for an exact measurement. You can change the settling time of the multiplexer with the instruction **SEQ_SET_DELAY**.

See also

SEQ_SELECT, SEQ_STATUS, SEQ_READ

To 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

```
#DEFINE module 1
#INCLUDE ADWPAD.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
                             'temporary register
                             'and copy them to DATA_1
```

SEQ_READ

The instruction **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 ADWPAD.INC
SEQ_READ(module,length,array[],array_idx)
```

Parameters

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

Notes

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

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

See also

SEQ_READ_ONE, SEQ_READ_TWO, SEQ_READ_PACKED, SEQ_READ32, SEQ_MODE, SEQ_READ

To be used for the modules

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 ADWPAD.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 the
                            'Continous 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 temporary register into DATA_1
```

SEQ_READ_ONE

The instruction **SEQ_READ_ONE** reads out a specified measurement value (16 bit) of a measurement group on the specified module.

Syntax

```
#INCLUDE ADWPAD.INC
ret_val = SEQ_READ_ONE(module,idxno)
```

Parameters

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

Notes

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

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

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

See also

SEQ_MODE, SEQ_SELECT, SEQ_STATUS, SEQ_READ

To be used for the modules

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

SEQ_READ_TWO

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

Syntax

```
#INCLUDE ADWPAD.INC
ret_val = SEQ_READ_TWO(module,idxno)
```

Parameters

`module` Specified module address — LONG

`idxno` Index number that indicates the position of 2 channels in the measurement group (0 ... 15). — LONG

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

`ret_val` 32-bit value that contains the 2 measurement values of the channels (indirectly selected by `idxno`). — LONG

Notes

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

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

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

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

See also

SEQ_MODE, SEQ_SELECT, SEQ_STATUS, SEQ_READ

Can be used for the modules

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

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

Example

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

SEQ_READ_PACKED

The instruction **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 ADWPAD.INC
SEQ_READ_PACKED(module,length,array[],array_idx)
```

Parameters

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

Notes

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

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

See also

SEQ_READ_ONE, SEQ_READ_TWO, SEQ_READ_PACKED, SEQ_READ32, SEQ_MODE, SEQ_READ

To be used for the modules

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 ADWPAD.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
                            'the AIN-32/.. module
  START_CONV(1)             'Start measurement sequences in the
                            'Continous 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
                            'temporary register
                            'and copy them to DATA_1
```

SEQ_READ32

The instruction **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 ADWPAD.INC
SEQ_READ32(module, array[], array_idx)
```

Parameters

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

Notes

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

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

If you need more than sixteen measurement values, this instruction is faster than **SEQ_READ**, although **SEQ_READ32** always transfers all 32 measurement values.

See also

SEQ_READ_ONE, SEQ_READ_TWO, SEQ_READ_PACKED, SEQ_READ32, SEQ_MODE, SEQ_READ

To be used for the modules

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 ADWPAD.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 the
                            'Continous Mode
  WAIT_EOC(1)               'Wait, until all indicated channels
are
                            'measured once

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

SEQ_SELECT

The instruction **SEQ_SELECT** determines the channels belonging to the measurement group, which will be converted by the sequential control on the specified module.

Syntax

```
#INCLUDE ADWPAD.INC
SEQ_SELECT(module, pattern)
```

Parameters

module	Specified module address	LONG
pattern	Bit pattern that determines the channels, whose values are to be read.	LONG

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

Notes

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

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

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

See also

SEQ_MODE, SEQ_STATUS, SEQ_READ

To 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 ADWPAD.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
  ...                       '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
                            'temporary register
                            'and copy them to DATA_1
```

SEQ_SET_DELAY

The instruction **SEQ_SET_DELAY** determines the settling time (waiting time between 2 measurements) of the sequential control on the specified module.

Syntax

```
#INCLUDE ADWPAD.INC
SEQ_SET_DELAY(module,time)
```

Parameters

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

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

Notes

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

The settling time is calculated according to the following formula:

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

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

See also

SEQ_MODE, SEQ_READ

To be used for the modules

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 ADWPAD.INC

DIM DATA_1[32] AS LONG AT DM_LOCAL

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

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

SEQ_STATUS

The instruction **SEQ_STATUS** determines how many channels of the measurement group are already converted and stored by the sequential control of the specified module.

Syntax

```
#INCLUDE ADWPAD.INC
ret_val = SEQ_STATUS(module)
```

Parameters

module	Specified module address	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 measurement group, defined by **SEQ_SELECT**.

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

See also

SEQ_MODE, SEQ_SELECT, SEQ_STATUS, SEQ_READ

To be used for the modules

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

SH_SETMODE

The instruction **SH_SETMODE** sets the mode of the sample and hold levels.

Syntax

```
#INCLUDE ADWPAD.INC
SH_SETMODE(module,mode)
```

Parameters

module	Specified module address	LONG
mode	Mode of the sample & hold levels: 0: Sample 1: Hold	LONG

Notes

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

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

To be used for the modules

Pro-LPSH-8-FI

Pro-LPSH-4-FI

Pro-LPSH-8

Example

```
#INCLUDE ADWPAD.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

The instruction **START_CONV** starts the A/D conversion on the specified module.

Syntax

```
#INCLUDE ADWPAD.INC
START_CONV(module)
```

Parameters

module	Specified module address	LONG

Notes

If the module is released with **SYNCENABLE** for synchronization, the instruction **SYNCALL** has the same function as in **START_CONV**.

See also

WAIT_EOC, READADC

To be used for the modules

Pro-AIn-8/12, Pro-AIn-8/12 Rev. B

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

Pro-AIn-32/12, Pro-AIn-32/12 Rev. B

Pro-AIn-32/16 Rev. B

Pro-AO-16/8-12

Example

```
#INCLUDE ADWPAD.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 looop, which can only be used in processes with high priority:

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

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

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

START_CONVF

The instruction **START_CONVF** starts the conversion of one or more F-ADCs of the specified module.

Syntax

```
#INCLUDE ADWPAD.INC
START_CONVF(module,adc_nr)
```

Parameters

module	Specified module address	LONG
adc_nr	Bit pattern that determines the ADCs whose conversion is to be started (see table).	LONG

Bit no.	31:8	7	6	5	4	3	2	1	0
Conversion no.	–	8	7	6	5	4	3	2	1

Notes

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

You can start a conversion with the instruction **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

To be used for the modules

Pro-AIn-F-4/12 Rev. A, Pro-AIn-F-8/12 Rev. A

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Pro-AIn-F-4/16 Rev. A, Pro-AIn-F-8/16 Rev. A

Example

```
#INCLUDE ADWPAD.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

The instruction **SYNC_MODE** determines on the specified module the type of synchronization with other modules, especially for burst-measurement sequences.

Syntax

```
#INCLUDE ADWPAD.INC
SYNC_MODE(module,mode)
```

Parameters

module	Specified module address	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

To be used for the modules

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Example

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

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

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

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

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

WAIT_EOC

The instruction **WAIT_EOC** waits, until the last A/D conversion has finished.

Syntax

```
#INCLUDE ADWPAD.INC
WAIT_EOC(module)
```

Parameters

module	Specified module address	LONG

See also

START_CONV, READADC

To be used for the modules

Pro-AIn-8/12, Pro-AIn-8/12 Rev. B

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

Pro-AIn-32/12, Pro-AIn-32/12 Rev. B

Pro-AIn-32/16 Rev. B

Pro-AO-16/8-12 Rev. A

Pro-LPSH-4-FI, Pro-LPSH-8-FI, Pro-LPSH-8

Example

```
#INCLUDE ADWPAD.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:

```
  time = READ_TIMER()       'Start AD conversion
  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

The instruction **WAIT_EOCF** waits until the end of conversion on all specified F-ADCs.

Syntax

```
#INCLUDE ADWPAD.INC
WAIT_EOCF(module,adc_no)
```

Parameters

module	Specified module address	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

START_CONVF, READADCF

To be used for the modules

Pro-AIn-F-4/12 Rev. A, Pro-AIn-F-8/12 Rev. A

Pro-AIn-F-4/14 Rev. A, Pro-AIn-F-8/14 Rev. A

Pro-AIn-F-4/16 Rev. A, Pro-AIn-F-8/16 Rev. A

Example

```
#INCLUDE ADWPAD.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
```

3.3 ADWPDA.INC

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

```
#INCLUDE ADWPDA.INC
```

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

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

In the list of modules you will find which of the functions corresponds to the *ADwin-Pro* modules.

Note: It is presumed that in all application examples on the D/A module the address is set to 1.

DAC

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

Syntax

```
#INCLUDE ADWPDA.INC
DAC(module,dac_no,value)
```

Parameters

module	Specified module address	LONG
dac_no	Number of the output	LONG
value	value to output (0…65.535)	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

To be used for the modules

Pro-AOut-4/16

Pro-AOut-8/16

Pro-AOut-16/8-12

Example

```
REM Digital proportionl controller
#INCLUDE ADWPAD.INC
#INCLUDE ADWPDA.INC
DIM sp, dev AS LONG     'Declaration
DIM g, actuate AS LONG  'Declaration

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

FG_CONTROL

The instruction **FG_CONTROL** starts or stops the function generator (output of values) on the selected output channels of the module.

Syntax

```
#INCLUDE ADWPDA.INC
FG_CONTROL(module,output,run_mode)
```

Parameters

module	Specified module address	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

To be used for the modules

Pro-AOut-4/16-M2 Rev. C

Example

```
#INCLUDE ADWPDA.INC

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

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

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

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

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

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

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

FG_DEF

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

Syntax

```
#INCLUDE ADWPDA.INC
FG_DEF(module,DACno,startadr,memsize)
```

Parameters

module	Specified module address	LONG
DACno	Numbers of the output channel (1…4)	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 1048575 (2^{20}-1) values à 16-bit each. The start address and the size of the buffer can individually be selected for each channel; there is no automatic check for range violations.

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

See also

FG_CONTROL, FG_DELAY, FG_MODE, FG_READ_INDEX, FG_STATUS, FG_WRITE

To be used for the modules

Pro-AOut-4/16-M2 Rev. C

Example

```
#INCLUDE ADWPDA.INC

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

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

FG_DELAY

The instruction **FG_DELAY** sets the output rate of function generator on the specified module.

Syntax

```
#INCLUDE ADWPDA.INC
FG_DELAY(module,DACno,scale)
```

Parameters

module	Specified module address	LONG
DACno	Numbers (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

To be used for the modules

Pro-AOut-4/16-M2 Rev. C

Beispiel

```
#INCLUDE ADWPDA.INC

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

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

FG_MODE

The instruction FG_MODE enables or disables the function generator mode on the specified module.

Syntax

```
#INCLUDE ADWPDA.INC
FG_MODE(module,mode)
```

Parameters

module	Specified module address	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 and, if possible, in 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

To be used for the modules

Pro-AOut-4/16-M2 Rev. C

Example

```
#INCLUDE ADWPDA.INC

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

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

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

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

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

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

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

FG_READ_INDEX

The instruction **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 ADWPDA.INC
ret_val = FG_READ_INDEX(module,DACno)
```

Parameters

module	Specified module address	LONG
DACno	Numbers (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

To be used for the modules

Pro-AOut-4/16-M2 Rev. C

Example

```
#INCLUDE ADWPDA.INC

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

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

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

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

FG_STATUS

The instruction **FG_STATUS** returns the status of all function generators of the module.

Syntax

```
#INCLUDE ADWPDA.INC
ret_val = FG_STATUS(module)
```

Parameters

module	Specified module address	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 the instruction **FG_CONTROL**(…,2) (soft stop), its bit in ret_val will only be reset to 0 (zero) when the last value in the range is output.

See also

FG_CONTROL, FG_DEF, FG_DELAY, FG_MODE, FG_READ_INDEX, FG_WRITE

To be used for the modules

Pro-AOut-4/16-M2 Rev. C

Example

```
#INCLUDE ADWPDA.INC

#DEFINE 1 1              'Address of Aout-module

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

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

EVENT:
  …

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

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

FG_WRITE

The instruction **FG_WRITE** transfers an even number of data of an array to a specified address in the buffer of the module.

Syntax

```
#INCLUDE ADWPDA.INC
FG_MODE(module,startadr,length,array[],array_idx)
```

Parameters

module	Specified module address	LONG
startadr	Start address (0…0FFFFFh) in the buffer. Beginning here, data are stored.	LONG
length	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	ARRAY 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 length must correspond to the settings made under **FG_DEF**.

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

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

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

See also

FG_CONTROL, FG_DEF, FG_DELAY, FG_MODE, FG_READ_INDEX, FG_STATUS

To be used for the modules

Pro-AOut-4/16-M2 Rev. C

Example

```
#INCLUDE ADWPDA.INC

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

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

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

EVENT:
  …
```

START_DAC

The instruction START_DAC starts the conversion or output of all DACs on the specified module.

Syntax

```
#INCLUDE ADWPDA.INC
START_DAC(module)
```

Parameters

module	Specified module address	LONG

See also

WRITEDAC, DAC

To be used for the modules

Pro-AOut-4/16

Pro-AOut-8/16

Pro-AOut-16/8-12

Example

```
REM Simultaneous output of two different signals
REM on the outputs 1 and 2 of a D/A module.
#INCLUDE ADWPDA.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

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

Syntax

```
#INCLUDE ADWPDA.INC
WRITEDAC(module,dac_no,value)
```

Parameters

module	Specified module address	LONG
dac_no	Number of the output	LONG
value	Value to output (0…4.095 with 12-bit DAC, 0…65.535 with 16-bit DAC)	LONG

See also

START_DAC, DAC

To be used for the modules

Pro-AOut-4/16

Pro-AOut-8/16

Pro-AOut-16/8-12

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 ADWPDA.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
```

3.4 ADWPDIO.INC

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

```
#INCLUDE ADWPDIO.INC
```

in your *ADbasic* program you can use all functions and procedures of this file.

On the following pages all functions of the file ADWPDIO.INC will be described more detailed. In addition an easy application example illustrates each function.

It is assumed that in all application examples the address on the digital module is set to 1.

CNT_CLEAR

The instruction **CNT_CLEAR** sets the counter values of one or more counters on the specified module to the value 0 (zero).

Syntax

```
#INCLUDE ADWPDIO.INC
CNT_CLEAR(module,pattern)
```

Parameters

module	Specified module address	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

To be used for the modules

Pro-CNT-16/16, Pro-CNT-16/16-I

Pro-CNT-8/32, Pro-CNT-8/32-I, Pro-CNT-16/32

Pro-CNT-VR4, Pro-CNT-VR4-L, Pro-CNT-VR4-I, Pro-CNT-VR4-L-I

Pro-CNT-VR2PW2, Pro-CNT-VR2PW2-I

Pro-CNT-PW4, Pro-CNT-PW4-I

Pro-CO4-T, Pro-CO4-I, Pro-CO4-D

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

```
#INCLUDE ADWPDIO.INC

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

CNT_ENABLE

The instruction **CNT_ENABLE** enables or disables one or more counters on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
CNT_ENABLE(module,pattern)
```

Parameters

module	Specified module address	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

To be used for the modules

Pro-CNT-16/16, Pro-CNT-16/16-I

Pro-CNT-8/32, Pro-CNT-8/32-I, Pro-CNT-16/32

Pro-CNT-VR4, Pro-CNT-VR4-L, Pro-CNT-VR4-I, Pro-CNT-VR4-L-I

Pro-CNT-VR2PW2, Pro-CNT-VR2PW2-I

Pro-CNT-PW4, Pro-CNT-PW4-I

Pro-CO4-T, Pro-CO4-I, Pro-CO4-D

Example

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

```
#INCLUDE ADWPDIO.INC

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

CNT_LATCH

The instruction **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 ADWPDIO.INC
CNT_LATCH(module,pattern)
```

Parameters

module	Specified module address	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

See also

CNT_READLATCH16, CNT_READLATCH32

To be used for the modules

Pro-CNT-16/16, Pro-CNT-16/16-I

Pro-CNT-8/32, Pro-CNT-8/32-I, Pro-CNT-16/32

Pro-CNT-VR4, Pro-CNT-VR4-I, Pro-CNT-VR4-L, Pro-CNT-VR4-L-I

Pro-CNT-VR2PW2, Pro-CNT-VR2PW2-I

Pro-CO4-T, Pro-CO4-I, Pro-CO4-D

Example

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

```
#INCLUDE ADWPDIO.INC
DIM value AS LONG

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

CNT_READ16

The instruction **CNT_READ16** returns the current counter value of a 16-bit counter on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = CNT_READ16(module,cnt_no)
```

Parameter

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

Example

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

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

For special applications you can also use these instructions instead of **CNT_READ16**.

See also

CNT_READ32, CNT_LATCH, CNT_READLATCH16

To be used for the modules

Pro-CNT-16/16, Pro-CNT-16/16-I

Example

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

CNT_READ32

The instruction **CNT_READ32** returns the current counter value of a 32-bit counter on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = CNT_READ32(module,cnt_no)
```

Parameters

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

Notes

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

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

To be used for the modules

Pro-CNT-8/32, Pro-CNT-8/32-I

Pro-CNT-VR4, Pro-CNT-VR4-L, Pro-CNT-VR4-I, Pro-CNT-VR4-L-I

Pro-CNT-VR2PW2, Pro-CNT-VR2PW2-I

Pro-CNT-PW4, Pro-CNT-PW4-I

Example

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

CNT_READLATCH16

The instruction **CNT_READLATCH16** returns the value from the latch register of a 16-bit counter on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = CNT_READLATCH16(module,cnt_no)
```

Parameters

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

Notes

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

See also

CNT_LATCH, CNT_READ16

To be used for the modules

Pro-CNT-16/16, Pro-CNT-16/16-I

Example

```
#INCLUDE ADWPDIO.INC
DIM value AS LONG

INIT:
  value = ADC(1,1)          'Get measurement value
  IF (value>49151) THEN CNT_LATCH(1,3)
                            'Latch counters 1 and 2
  PAR_1 = CNT_READLATCH16(1,1) - CNT_READLATCH16(1,2)
                            'Difference counters 1 and 2
```

CNT_READLATCH32

The instruction **CNT_READLATCH32** returns the value from the latch register of a 32-bit counter on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = CNT_READLATCH32(module,cnt_no)
```

Parameters

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

Notes

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

See also

CNT_LATCH, CNT_READ32

To be used for the modules

Pro-CNT-8/32, Pro-CNT-8/32-I

Pro-CNT-VR4, Pro-CNT-VR4-L, Pro-CNT-VR4-I, Pro-CNT-VR4-L-I

Pro-CNT-VR2PW2, Pro-CNT-VR2PW2-I

Pro-CNT-PW4, Pro-CNT-VR4-I

Example

```
#INCLUDE ADWPDIO.INC
DIM value AS LONG

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

CNT_SETMODE

The instruction **CNT_SETMODE** sets the operating mode of all counters on the specified module, four edge evaluation or clock and direction input.

Syntax

```
#INCLUDE ADWPDIO.INC
CNT_SETMODE(module,pattern)
```

Parameters

module	Specified module address	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.

To be used for the modules

Pro-CNT-VR4, Pro-CNT-VR4-L, Pro-CNT-VR4-I, Pro-CNT-VR4-L-I

Pro-CNT-VR2PW2, Pro-CNT-VR2PW2-I

Example

```
#INCLUDE ADWPDIO.INC

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

CO4_ CLEARENABLE

The instruction **CO4_CLEARENABLE** enables the external input CLR of one or more counters.

Syntax

```
#INCLUDE ADWPDIO.INC
CO4_CLEARENABLE(module,pattern)
```

Parameters

module	Specified module address	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".

Note:

The external input may be enabled either with the instruction **CO4_CLEARENABLE** or with **CO4_LATCHENABLE**, but not with both instructions simultaneously!

See also

CO4_LATCHENABLE

To be used for the modules

Pro-CO4-D, Pro-CO4-I, Pro-CO4-T

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  CO4_SETMODE(1,1,1)        'Counter 1: CLK/DIR mode
  CO4_LATCHENABLE(1,0)      'Disable latch-input for all counters
  CO4_CLEARENABLE(1,1)      'Enable clear-input for counter 1
  CNT_CLEAR(1,1)            'Clear counter 1
  CNT_ENABLE(1,1)           'Start counter 1
EVENT:
  PAR_1 = CO4_READ(1,1)     'Read out counter 1
```

CO4_GETSTATUS

The instruction **CO4_GETSTATUS** returns the status of the input signals of a counter on the specified module as bit pattern.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = CO4_GETSTATUS(module,cnt_no)
```

Parameters

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

Bit no.						
6	5	4	3	2	1	0
Current status of a signal input (*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 the instruction **CO4_RESETSTATUS** to delete the message.
Bit 0 = 1: Cable error; the differential signals (A & /A or B & /B or C & /C; C = CLR-/LATCH) have one of the following errors:
A cable is not connected (cable break), short circuit, signal voltage is too low, common-mode voltage is too high or slew rate is too low (< 0.33 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 the instruction **CO4_CLEARENABLE**).
Bit 3 = 1: LATCH signal exists (when it has been released by the instruction **CO4_LATCHENABLE**).

Notes

A cable error (bit 0) can only be detected at differential inputs! At TTL inputs these bits are always 0 (zero) and the instruction **CO4_RESETSTATUS** has no effect.

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

See also

CO4_RESETSTATUS

To be used for the modules

Pro-CO4-D, Pro-CO4-I, Pro-CO4-T

Example

```
#INCLUDE ADWPDIO.INC
DIM error AS LONG

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

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

CO4_ LATCHENABLE

The instruction **CO4_LATCHENABLE** enables the external input LATCH of one or more counters on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
CO4_LATCHENABLE(module,pattern)
```

Parameters

module	Specified module address	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 the instruction **CO4_CLEARENABLE** or with **CO4_LATCHENABLE**, but not with both instructions simultaneously!

See also

CO4_CLEARENABLE

To be used for the modules

Pro-CO4-D, Pro-CO4-I, Pro-CO4-T

Example

```
#INCLUDE ADWPDIO.INC

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

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

CO4_READ

The instruction **CO4_READ** returns the current counter value from the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = CO4_READ(module,cnt_no)
```

Parameters

module	Specified module address	LONG
cnt_no	Counter number	LONG
ret_val	Current counter value of the specified counter	LONG

Notes

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

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

To be used for the modules

Pro-CNT-16/32,

Pro-CO4-D, Pro-CO4-I, Pro-CO4-T

Example

```
#INCLUDE ADWPDIO.INC

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

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

CO4_READLATCH

The instruction **CO4_READLATCH** returns the value of the latch register of a counter on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = CO4_READLATCH(module,cnt_no)
```

Parameters

module	Specified module address	LONG
cnt_no	Counter number	LONG
ret_val	Value from the latch register of the counter	LONG

Notes

In order to get the current counter value, it has to be latched before into the latch register with the instruction **CNT_LATCH** and can then be read by **CO4_READLATCH**.

See also

CNT_LATCH, CO4_READ

To be used for the modules

Pro-CNT-16/32,

Pro-CO4-D, Pro-CO4-I, Pro-CO4-T

Example

```
#INCLUDE ADWPDIO.INC
DIM old, new AS LONG        'Dimensioning the variables

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

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

CO4_ RESETSTATUS

The instruction CO4_RESETSTATUS clears the status register of one or more counters on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
CO4_RESETSTATUS(module,pattern)
```

Parameters

module	Specified module address	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

See also

CO4_GETSTATUS

To be used for the modules

Pro-CO4-D, Pro-CO4-I, Pro-CO4-T

Example

```
#INCLUDE ADWPDIO.INC
DIM error AS LONG

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

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

CO4_SET_LATCHMODE

The instruction **CO4_SET_LATCHMODE** determines the mode of the latch-inputs for all counters on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
CO4_SET_LATCHMODE(module,pattern)
```

Parameters

module	Specified module address	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.

The instruction **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

To be used for the modules

Pro-CO4-D, Pro-CO4-I, Pro-CO4-T

CO4_SETMODE

The instruction **CO4_SETMODE** sets the count mode of a counter on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
CO4_SETMODE(module,cnt_no,mode)
```

Parameters

module	Specified module address	LONG
cnt_no	Counter number	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

To be used for the modules

Pro-CO4-D, Pro-CO4-I, Pro-CO4-T

Example

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

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

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

DIG_LATCH

The instruction **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 ADWPDIO.INC
DIG_LATCH(module)
```

Parameters

module	Specified module address	LONG

Notes

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

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

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

If the module is released for synchronization by **SYNCENABLE**, the instruction **SYNCALL** has the same functions as the instruction **DIG_LATCH**.

See also

DIG_READLATCH1, DIG_READLATCH2, DIG_WRITELATCH1, DIG_WRITELATCH2, DIG_WRITELATCH32, EXTLCH_ENABLE

DIGPROG1, DIGPROG2, DIGIN_WORD1, DIGIN_WORD2, DIGOUT, DIGOUT_WORD1, DIGOUT_WORD2

DIGIN_LONG_F, DIGOUT_BITS_F, DIGOUT_F, DIGOUT_LONG_F

GET_DIGOUT_LONG,
GET_DIGOUT_WORD1, GET_DIGOUT_WORD2

SYNCALL, SYNCENABLE

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Pro-OPT-16, Pro-REL-16, Pro-TRA-16

Example

```
#INCLUDE ADWPDIO.INC

INIT:
DIGPROG1(1,0FFFFh)          'DIO15:00 (low-word) of the DIO-32-
                            'module as output
DIGPROG2(1,0)               'DIO31:16 (high-word) of the DIO-32-
                            'module as input
DIG_WRITELATCH1(1,0)        'Set all output bits to 0

EVENT:
DIG_LATCH(1)                'Latch inputs, output contents of the
                            'output latch
…                           'more program steps
PAR_1 = DIG_READLATCH2(1)   'Read in high-word and output at …
DIG_WRITELATCH1(1,PAR_1)    'the next event in the low-word.
```

DIG_READLATCH1

The instruction **DIG_READLATCH1** returns the lower 16 bits (bit 0…bit 15) from the latch register for the digital inputs of the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
DIG_READLATCH1(module)
```

Parameters

module	Specified module address	LONG

Notes

It is recommended to first program the specified channels as inputs using the instruction **DIGPROG1**.

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

- DIGIN_WORD1
- DIGIN_WORD2
- SYNCALL (when activated)

See also

DIG_LATCH, DIG_READLATCH2, DIG_WRITELATCH1, DIG_WRITELATCH2, DIG_WRITELATCH32, EXTLCH_ENABLE

DIGPROG1, DIGPROG2, DIGIN_WORD1, DIGIN_WORD2, DIGIN_LONG_F

SYNCALL, SYNCENABLE

To be used for the modules

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

Example

```
#INCLUDE ADWPDIO.INC
#INCLUDE ADWINPRO.INC
DIM value AS LONG

INIT:
  REM Set DIO15:00 of modules 1+2 as inputs
  DIGPROG1(1,0)
  DIGPROG1(2,0)
  SYNCENABLE(1,dio,1)       'Enable synchronization of module 1
  SYNCENABLE(2,dio,1)       'Enable synchronization of module 2

EVENT:
  SYNCALL()                 'Transfer the logic level at the
                            'digital inputs of both modules
                            'synchronously to the latch register
  PAR_1 = DIG_READLATCH1(1) 'Read out temp. register of module 1
                            '(bits 0…15)
  PAR_2 = DIG_READLATCH1(2) 'Read out temp. register of module 2
                            '(bits 0…15)
```

DIG_READLATCH2

The instruction **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 ADWPDIO.INC
DIG_READLATCH2(module)
```

Parameters

module	Specified module address	LONG

Notes

It is recommended to first program the specified channels as inputs using the instruction **DIGPROG2**.

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

- DIGIN_WORD1
- DIGIN_WORD2
- SYNCALL (when activated)

See also

DIG_LATCH, DIG_READLATCH1, DIG_WRITELATCH1, DIG_WRITELATCH2, DIG_WRITELATCH32, EXTLCH_ENABLE

DIGPROG1, DIGPROG2, DIGIN_WORD1, DIGIN_WORD2, DIGIN_LONG_F

SYNCALL, SYNCENABLE

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC
#INCLUDE ADWINPRO.INC
DIM value AS LONG

INIT:
  REM Set DIO31:16 of modules 1+2 as inputs
  DIGPROG2(1,0)
  DIGPROG2(2,0)
  SYNCENABLE(1,dio,1)       'Enable synchronization of module 1
  SYNCENABLE(2,dio,1)       'Enable synchronization of module 2

EVENT:
  SYNCALL()                 'Transfer the logic level at the
                            'digital inputs of both modules
                            'synchronously to the latch register
  PAR_1 = DIG_READLATCH2(1) 'Read out temp. register of module 1
                            '(bits 16…31)
  PAR_2 = DIG_READLATCH2(2) 'Read out temp. register of module 2
                            '(bits 16…31)
```

DIG_ WRITELATCH1

The instruction **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 ADWPDIO.INC
DIG_WRITELATCH1(module,pattern)
```

Parameters

module	Specified module address	LONG
pattern	Bit pattern. Each bit corresponds to a digital output (see table).	LONG

Bit no.	31:16	15	14	13	…	2	1	0
Output no.	–	15	14	13	…	2	1	0

Notes

For the modules Pro-DIO-32 and Pro-DIO-32 Rev. B, the specified channels must be first programmed as outputs using the instruction **DIGPROG1**.

You can set the value of the latch register for the digital outputs with the following instructions:

- DIGOUT
- DIGOUT_WORD1
- DIGOUT_WORD2
- DIG_WRITELATCH1
- DIG_WRITELATCH2
- DIG_WRITELATCH32

The instruction must not be used in combination with **DIG_WRITELATCH2**. If you want to change bits both in the low and in the high word of the latch register, please use the instruction **DIG_WRITELATCH32**.

See also

DIG_LATCH, DIG_READLATCH1, DIG_WRITELATCH2, DIG_WRITELATCH32, EXTLCH_ENABLE

DIGPROG1, DIGPROG2, DIGOUT, DIGOUT_WORD1, DIGOUT_WORD2, GET_DIGOUT_WORD1, GET_DIGOUT_WORD2

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Pro-REL-16, Pro-TRA-16

Example

```
#INCLUDE ADWINPRO.INC
#INCLUDE ADWPDIO.INC
#INCLUDE ADWPAD.INC
DIM value AS LONG

INIT:
  REM DIO-32 only: Set DIO15:00 of the module as outputs
  DIGPROG1(1,0FFFFh)
  DIGPROG1(2,0FFFFh)

  SYNCENABLE(1,dio,1)       'Enable synchronization of digital
                            'module 1
  SYNCENABLE(2,dio,1)       '… and digital module 2
  DIG_WRITELATCH1(1,1)      'Set lowest bit in the output
                            'latch register
  DIG_WRITELATCH1(2,1)      'Set lowest bit in the output
                            'latch register

EVENT:
  value = ADC(1,1)          'Measurement data acquisition
  IF (value > 3000) THEN    'Limit value exceeded?
    SYNCALL()               'Output values from the latch
                            'registers
  ENDIF
```

DIG_ WRITELATCH2

The instruction **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 ADWPDIO.INC
DIG_WRITELATCH2(module,pattern)
```

Parameters

module	Specified module address	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 the instruction **DIGPROG2**.

You can set the value of the latch register for the digital outputs with the following instructions:

- DIGOUT
- DIGOUT_WORD1
- DIGOUT_WORD2
- DIG_WRITELATCH1
- DIG_WRITELATCH2
- DIG_WRITELATCH32

You must not use this 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 the instruction **DIG_WRITELATCH32**.

See also

DIG_LATCH, DIG_READLATCH1, DIG_WRITELATCH1, DIG_WRITELATCH32, EXTLCH_ENABLE

DIGPROG1, DIGPROG2, DIGOUT, DIGOUT_WORD1, DIGOUT_WORD2, GET_DIGOUT_WORD1, GET_DIGOUT_WORD2

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWINPRO.INC
#INCLUDE ADWPDIO.INC
#INCLUDE ADWPAD.INC
DIM value AS LONG

INIT:
  REM DIO-32 only: Set DIO31:16 of the module as outputs
  DIGPROG2(1,0FFFFh)
  DIGPROG2(2,0FFFFh)

  SYNCENABLE(1,dio,1)       'Enable synchronization of module 1
  SYNCENABLE(2,dio,1)       '… and module 2
  DIG_WRITELATCH2(1,1)      'Set bit 16 in the output latch
                            'register
  DIG_WRITELATCH2(2,10000b)'Set bit 20 in the output latch
                            'register

EVENT:
  value = ADC(1,1)          'Measurement data acquisition
  IF (value > 3000) THEN    'Limit value exceeded?
    SYNCALL()               'Output values from the latch
                            'registers
  ENDIF
```

DIG_ WRITELATCH32

The instruction **DIG_WRITELATCH32** writes a 32-bit value into the long-word (bits 31…0) of the latch on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
DIG_WRITELATCH32(module,pattern)
```

Parameters

module	Specified module address	LONG
pattern	Bit pattern. Each bit corresponds to a digital output (see table).	LONG

Bit no.	31	30	29	…	2	1	0
Output no.	31	30	29	…	18	17	16

Notes

The specified channels must be first programmed as outputs using the instructions **DIGPROG1** and **DIGPROG2**.

You can set the value of the latch register for the digital outputs with the following instructions:

- DIGOUT
- DIGOUT_WORD1
- DIGOUT_WORD2
- DIG_WRITELATCH1
- DIG_WRITELATCH2
- DIG_WRITELATCH32

See also

DIG_LATCH, DIG_READLATCH1, DIG_WRITELATCH1, DIG_ WRITELATCH2, EXTLCH_ENABLE

DIGOUT, DIGOUT_WORD1, DIGOUT_WORD2, DIGPROG1, DIGPROG2, GET_DIGOUT_WORD1, GET_DIGOUT_WORD2

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC

INIT:
DIGPROG1(1,0FFFFh)          'DIO15:00 (low-word) of the DIO-32-
                            'module as output
DIGPROG2(1,0FFFFh)          'DIO31:16 (high-word) of the DIO-32-
                            'module as output

EVENT:
DIG_LATCH(1)                'Output information of the output
latch
                            'on a DIO-32 board
DIG_WRITELATCH32(1,PAR_1)   'Write long-word into output latch
```

DIGIN_LONG_F

The instruction **DIGIN_LONG_F** returns the status of the inputs (bits 31…00) of the specified module as bit pattern.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = DIGIN_LONG_F(module)
```

Parameters

module	Specified module address	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

It is recommended to first program the specified channels as inputs using the instructions **DIGPROG1** and **DIGPROG2**.

See also

DIG_LATCH, DIGIN_WORD1, DIGIN_WORD2, DIGPROG1, DIGPROG2

DIGOUT_LONG_F, DIGOUT_WORD1, DIGOUT_WORD2

To be used for the modules

Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC

INIT:
DIGPROG1(1,0)               'Set DIO15:00 (Low-Word) as inputs
DIGPROG2(1,0)               'Set DIO31:16 (High-Word) as inputs

EVENT:
  PAR_1 = DIGIN_LONG_F(1)   'Read all inputs
```

DIGIN_WORD1

The instruction **DIGIN_WORD1** returns the status of the inputs 0…15 of the specified module as bit pattern.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = DIGIN_WORD1(module)
```

Parameters

module	Specified module address	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

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

See also

DIGIN_WORD2, DIGOUT, DIGOUT_WORD1, DIGOUT_WORD2, DIGPROG1, DIGPROG2

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Pro-OPT-16

Example

```
#INCLUDE ADWPDIO.INC
#INCLUDE ADWPAD.INC
DIM DATA_1[10000] AS LONG AS FIFO

INIT:
  REM DIO32 only: Set DIO15:00 (Low-Word) as inputs
  DIGPROG1(4,0)

EVENT:
  IF (DIGIN_WORD1(4) AND 14 = 14) THEN
                          'Query, if inputs 1, 2 and 3
                          'on mdoule 4 are set
    DATA_1 = ADC(1,1)     'Measurement data acquisition
  ENDIF
```

The bit pattern of the decimal value 14 (which is …01110b) enables you to recognize which digital inputs are set, here the inputs 1, 2 and 3.

In *ADbasic* you can also enter numbers in binary notation. Please add the letter "b" to the bit pattern. The line in the example above would then be as follows:

```
IF (DIGIN_WORD1(4) AND 1110b = 1110b) THEN
```

DIGIN_WORD2

The instruction **DIGIN_WORD2** returns the status of the inputs 16…31 of the specified module as bit pattern.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = DIGIN_WORD2(module)
```

Parameters

module	Specified module address	LONG
ret_val	Bit pattern. Each of the bits 15…0 corresponds to the input status of a digital input (see table). Bit = 0: Low level Bit = 1: High level	LONG

Bit no.	31:16	15	14	…	2	1	0
Input	–	31	30	…	18	17	16

Notes

It is recommended to first program the specified channels as inputs using the instruction **DIGPROG2**.

See also

DIGIN_WORD1, DIGOUT, DIGOUT_WORD1, DIGOUT_WORD2, DIGPROG1, DIGPROG2

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC
#INCLUDE ADWPAD.INC
DIM DATA_1[10000] AS LONG AS FIFO

INIT:
  DIGPROG2(4,0)              'DIO31:16 (high word) as inputs

EVENT:
   'Query if inputs 16, 18 and 21 are set on module 4
  IF (DIGIN_WORD2(4) AND 39 = 39) THEN
    DATA_1 = ADC(1,1)        'Measurement data acquisition
  ENDIF
```

The bit pattern of the decimal value 39 (which is …0100101b) enables you to recognize which digital inputs are set, here the inputs 16, 18 and 21.

In *ADbasic* you can also enter numbers in binary notation. Please add the letter "b" to the bit pattern. The line in the example above would then be as follows:

```
IF (DIGIN_WORD2(4) AND 0100101b = 0100101b) THEN
```

DIGOUT

The instruction **DIGOUT** sets a single output of the specified module to the level "high" or "low". All other outputs remain unchanged.

Syntax

```
#INCLUDE ADWPDIO.INC
DIGOUT(module,output,value)
```

Parameters

module	Specified module address	LONG
output	Number of the output to be set (0…31)	LONG
value	New status of the selected output: 0: Low level 1: High level	LONG

Notes

The instruction can also be used for the module Pro-DIO-32 Rev. B. But we recommend to use the instruction **DIGOUT_F**, because it works faster and needs essentially less program memory.

The specified channel must be first programmed as output using the instruction **DIGPROG1** or **DIGPROG2**.

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

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

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Pro-REL-16, Pro-TRA-16

Example

```
#INCLUDE ADWPDIO.INC
#INCLUDE ADWPAD.INC
DIM value AS LONG

INIT:
  REM DIO32 only: Set channels as outputs
  DIGPROG1(1,0FFFFh)        'DIO15:00 (low word) as outputs
  DIGPROG2(1,0FFFFh)        'DIO31:16 (high word) as outputs

EVENT:
  value = ADC(1,1)          'Measurement data acquisition
  IF (value < 100) THEN     'If value is below limit
    DIGOUT(1,2,0)           'reset digital output 2
  ENDIF
```

DIGOUT_BITS_F

The instruction **DIGOUT_BITS_F** sets the specified outputs of the specified module to the levels "high" or "low".

Syntax

```
#INCLUDE ADWPDIO.INC
DIGOUT_BITS_F(module,set,clear)
```

Parameters

module	Specified module address	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 ouputs to the level "low". Bit = 0: Output remains unchanged. Bit = 1: Set output to level "low"	LONG

Bit no.	31	30	…	2	1	0
Output	31	30	…	2	1	0

Notes

The specified channels must be first programmed as outputs using the instructions **DIGPROG1** and **DIGPROG2**.

You can set or clear any required outputs without changing the status of the remaining outputs. The logical combination of the corresponding registers is made with this instruction on hardware level. Thus it runs much faster than the instruction **DIGOUT**, that works on software level.

For clarity reasons please note that the bits in `set` must not be set in the bit pattern `clear` at the same time, and vice versa.

See also

DIGOUT, DIGOUT_F, DIGOUT_WORD1, DIGOUT_WORD2, DIGPROG1, DIGPROG2

Can also be use for the modules

Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  REM DIO32 only: Set channels as outputs
  DIGPROG1(1,0FFFFh)        'DIO15:00 (low word) als Ausgang
  DIGPROG2(1,0FFFFh)        'DIO31:16 (high word) als Ausgang

EVENT:
  IF (PAR_1 = 1) THEN       'Get condition
    DIGOUT_BITS_F(1,8888h,7777h)'Set MSB of every byte,
                            'Clear all other bits
  ELSE
    DIGOUT_BITS_F(1,5555h,0AAAAh)'Set odd-numbered bits,
                            'clear even-numbered bits
  ENDIF
```

DIGOUT_F

The instruction **DIGOUT_F** sets a single output of the specified module to the level "high" or "low". All other outputs remain unchanged.

Syntax

```
#INCLUDE ADWPDIO.INC
DIGOUT_F(module,output,value)
```

Parameters

module	Specified module address	LONG
output	Number of the output to be set (0…31)	LONG
value	New status of the selected output: 0: Low level 1: High level	LONG

Notes

The specified channels must be first programmed as outputs using the instructions **DIGPROG1** and **DIGPROG2**.

The logical combination of the corresponding registers is made with this instruction on hardware level. Thus it runs much faster than the instruction **DIGOUT**, that works on software level.

See also

DIGOUT, DIGOUT_BITS_F, DIGOUT_WORD1, DIGOUT_WORD2, DIGPROG1, DIGPROG2

To be used for the modules

Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  REM DIO32 only: Set channels as outputs
  DIGPROG1(1,0FFFFh)        'DIO15:00 (low word) as outputs
  DIGPROG2(1,0FFFFh)        'DIO31:16 (high word) as outputs

EVENT:
  IF (DIGIN_WORD1(1) AND 8000h = 1) THEN
                            'Read low-word (bits 0…15) and
                            'check, if MSB is set
    DIGOUT_F(2,31,0)        'If MSB is set, clear bit 31
  ELSE
    DIGOUT_F(2,31,1)        'If MSB is cleared, set bit 31
  ENDIF
```

DIGOUT_LONG_F

With the 32-bit value the instruction **DIGOUT_LONG_F** sets or clears all outputs on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
DIGOUT_LONG_F(module,pattern)
```

Parameters

module	Specified module address	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

To be used for the modules

Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  DIGPROG1(1,0FFFFh)        'DIO15:00 (low word) as outputs
  DIGPROG2(1,0FFFFh)        'DIO31:16 (high word) as outputs

EVENT:
  DIGOUT_LONG_F(1,1000000) 'Output the value 1 million as binary
                           'value on the DIOs
```

DIGOUT_WORD1

The instruction **DIGOUT_WORD1** sets the digital outputs 0…15 on the specified module simultaneously to the specified levels.

Syntax

```
#INCLUDE ADWPDIO.INC
DIGOUT_WORD1(module, pattern)
```

Parameters

module	Specified module address	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 the instruction **DIGPROG1**.

See also

DIG_LATCH, DIG_READLATCH1, DIG_READLATCH2, DIG_WRITELATCH1, DIG_WRITELATCH2, DIG_WRITELATCH32, EXTLCH_ENABLE

DIGPROG1, DIGPROG2, DIGIN_WORD1, DIGIN_WORD2, DIGOUT, DIGOUT_WORD2

DIGIN_LONG_F, DIGOUT_BITS_F, DIGOUT_F, DIGOUT_LONG_F

GET_DIGOUT_LONG, GET_DIGOUT_WORD1, GET_DIGOUT_WORD2

See also

DIGIN_WORD1

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Pro-REL-16, Pro-TRA-16

Example

```
#INCLUDE ADWPDIO.INC
#INCLUDE ADWPAD.INC
DIM value AS LONG

INIT:
  REM DIO32 only: Set channels as outputs
  DIGPROG1(4,0FFFFh)        'DIO15:00 (low word) as outputs

EVENT:
  value = ADC(1,1)          'Measurement data acquisition
  IF (value > 3000) THEN    'Is limit value exceeded?
    DIGOUT_WORD1(4,111b)    'Set outputs 0, 1 and 2 of the module 4
                            'all other outputs are reset.
  ENDIF
```

DIGOUT_WORD2

The instruction DIGOUT_WORD2 sets the digital outputs 16…31 on the specified module simultaneously to the specified levels.

Syntax

```
#INCLUDE ADWPDIO.INC
DIGOUT_WORD2(module,pattern)
```

Parameters

module	Specified module address	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 the instruction DIGPROG2.

See also

DIG_LATCH, DIG_READLATCH1, DIG_READLATCH2, DIG_WRITELATCH1, DIG_WRITELATCH2, DIG_WRITELATCH32, EXTLCH_ENABLE

DIGPROG1, DIGPROG2, DIGIN_WORD1, DIGIN_WORD2, DIGOUT, DIGOUT_WORD1

DIGIN_LONG_F, DIGOUT_BITS_F, DIGOUT_F, DIGOUT_LONG_F

GET_DIGOUT_LONG,
GET_DIGOUT_WORD1, GET_DIGOUT_WORD2

See also

DIGIN_WORD2

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC
#INCLUDE ADWPAD.INC
DIM value AS LONG

INIT:
  DIGPROG2(1,0FFFFh)          'DIO31:16 (high word) as outputs

EVENT:
  value = ADC(1,1)            'Measurement data acquisition
  IF (value > 3000) THEN      'Is limit value exceeded?
    DIGOUT_WORD2(4,1011b)     'set outputs 16, 17 a. 19, all
                              'other outputs are reset!
  ENDIF
```

DIGPROG1

The instruction DIGPROG1 programs the digital channels 0…15 of the specified module as input or output.

Syntax

```
#INCLUDE ADWPDIO.INC
DIGPROG1(module,pattern)
```

Parameters

module	Specified module address	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

Bit no.	Module	31…16	15	…	8	7	…	0
channel no.	DIO-32 RevA	–	15	…	08	07	…	00
	DIO-32 RevB	–	–	–	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 RevA: Each of the channels can be individually set as input or output.
- Pro-DIO-32 RevB: The channels can only be set as inputs or outputs in groups of 8 (2 relevant bits only, the other bits are ignored).

See also

DIG_LATCH, DIG_READLATCH1, DIG_READLATCH2, DIG_WRITELATCH1, DIG_WRITELATCH2, DIG_WRITELATCH32, EXTLCH_ENABLE

DIGPROG2, DIGIN_WORD1, DIGIN_WORD2, DIGOUT, DIGOUT_WORD1, DIGOUT_WORD2

DIGIN_LONG_F, DIGOUT_BITS_F, DIGOUT_F, DIGOUT_LONG_F

GET_DIGOUT_LONG,
GET_DIGOUT_WORD1, GET_DIGOUT_WORD2

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  DIGPROG1(1, 11111111b)    'Configures channels 0…7 of the DIO
                            'module no. 1 as outputs and channels
                            '8…15 as inputs
```

DIGPROG2

The instruction **DIGPROG2** programs all channels 16…31 of the specified module as inputs or outputs.

Syntax

```
#INCLUDE ADWPDIO.INC
DIGPROG2(module,pattern)
```

Parameters

module	Specified module address	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

Bit no.	Module	31…16	15	…	8	7	…	0
Channel no.	DIO-32 RevA	–	31	…	24	23	…	16
	DIO-32 RevB	–	–	–	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 RevA: Each of the channels can be individually set as input or output.
- Pro-DIO-32 RevB: The channels can only be set as inputs or outputs in groups of 8 (2 relevant bits only, the other bits are ignored).

See also

DIG_LATCH, DIG_READLATCH1, DIG_READLATCH2, DIG_WRITELATCH1, DIG_WRITELATCH2, DIG_WRITELATCH32, EXTLCH_ENABLE

DIGPROG1, DIGIN_WORD1, DIGIN_WORD2, DIGOUT, DIGOUT_WORD1, DIGOUT_WORD2

DIGIN_LONG_F, DIGOUT_BITS_F, DIGOUT_F, DIGOUT_LONG_F

GET_DIGOUT_LONG,
GET_DIGOUT_WORD1, GET_DIGOUT_WORD2

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  DIGPROG2(1, 11111111b)    'Configures channels 16…23 of the
                            'DIO module no. 1 as outputs and
                            ' channels 24…31 as inputs
```

EXTLCH_ENABLE

The instruction **EXTLCH_ENABLE** enables or disables all latch-inputs on the specified module. The latch-inputs are selected by the corresponding counter number.

Syntax

```
#INCLUDE ADWPDIO.INC
EXTLCH_ENABLE(module,cnt_select)
```

Parameters

module	Specified module address	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

In the folder <C:\ADwin\ADbasic\samples_ADwin_PRO> there is an example program <Pro-CNT-VR4-L-I_CLK-DIR.BAS>.

See also

DIG_LATCH, DIG_READLATCH1, DIG_READLATCH2, DIG_WRITELATCH1, DIG_WRITELATCH2, DIG_WRITELATCH32

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

To be used for the modules

Pro-CNT-VR4(-I), Pro-CNT-VR4-L(-I), Pro-CNT-VR2PW2

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  EXTLCH_ENABLE(1, 0011b)   'Enable latch-inputs of counters 1+2,
                            'disable latch-inputs of counters 3+4
```

GET_DIGOUT_LONG

The instruction **GET_DIGOUT_LONG** returns the contents of the output-latch (register for digital outputs) on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = GET_DIGOUT_LONG(module)
```

Parameters

module	Specified module address	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

Can also

Pro-DIO-32, Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC

EVENT:
PAR_1 = GET_DIGOUT_LONG(1) 'return bits 31:00 from the latch
```

GET_DIGOUT_WORD1

The instruction **GET_DIGOUT_WORD1** returns the lower word (bits 0…15) of the output-latch (register for digital outputs) on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = GET_DIGOUT_WORD1(module)
```

Parameters

module	Specified module address	LONG
ret_val	Contents of the output-latch (bits 15:00)	LONG

Notes

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

See also

DIG_LATCH, DIG_READLATCH1, DIG_READLATCH2, DIG_WRITELATCH1, DIG_WRITELATCH2, DIG_WRITELATCH32, EXTLCH_ENABLE

DIGPROG1, DIGPROG2, DIGIN_WORD1, DIGIN_WORD2, DIGOUT, DIGOUT_WORD1, DIGOUT_WORD2

DIGIN_LONG_F, DIGOUT_BITS_F, DIGOUT_F, DIGOUT_LONG_F

GET_DIGOUT_LONG, GET_DIGOUT_WORD2

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Pro-REL-16

Pro-TRA-16

Example

```
#INCLUDE ADWPDIO.INC
DIM value AS LONG

INIT:
  value = GET_DIGOUT_WORD1(1)'Read current value of the output
                             'register
  value = value AND 0FFFEh 'Set LSB (bit 0) to 0
  DIGOUT_WORD1(value)      'Return changed value
```

GET_DIGOUT_WORD2

The instruction **GET_DIGOUT_WORD2** returns the upper word (bits 16…31) of the output-latch (register for digital outputs) of the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = GET_DIGOUT_WORD2(module)
```

Parameters

module	Specified module address	LONG
ret_val	Contents of the output-latch (bits 31:16)	LONG

Notes

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

See also

DIG_LATCH, DIG_READLATCH1, DIG_READLATCH2, DIG_WRITELATCH1, DIG_WRITELATCH2, DIG_WRITELATCH32, EXTLCH_ENABLE

DIGPROG1, DIGPROG2, DIGIN_WORD1, DIGIN_WORD2, DIGOUT, DIGOUT_WORD1, DIGOUT_WORD2

DIGIN_LONG_F, DIGOUT_BITS_F, DIGOUT_F, DIGOUT_LONG_F

GET_DIGOUT_LONG, GET_DIGOUT_WORD1

To be used for the modules

Pro-DIO-32, Pro-DIO-32 Rev. B

Example

```
#INCLUDE ADWPDIO.INC
DIM value AS LONG

INIT:
  value = GET_DIGOUT_WORD2(1)'Read current value of the output
                             'register
  value = value AND 0FFFDh 'Set bit 17 to 0
  DIGOUT_WORD2(value)      'Return changed value
```

PWM_ENABLE

The instruction **PWM_ENABLE** enables or disables all internal counters of the specified module. The counters are selected by the corresponding PWM output number.

Syntax

```
#INCLUDE ADWPDIO.INC
PWM_ENABLE(module,output)
```

Parameters

module	Specified module address	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

This instruction *does not* change the level of the PWM outputs.
If you want to change the level of the PWM outputs, use the instruction **PWM_OUT** and afterwards stop the corresponding internal counters with **PWM_ENABLE**.

As soon as and as long as the counters are stopped, the PWM outputs cannot be changed.

See also

PWM_OUT, PWM_SET

To be used for the modules

Pro-PWM-4, Pro-PWM-4-I

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  PWM_ENABLE(1,1)           'Enable PWM output 1 for outputting
  PWM_SET(1,1,0,2500,2500) 'Set PWM signal for output 1
```

PWM_OUT

The instruction **PWM_OUT** sets a specified PWM output channel on the specified module to the level "high" or "low".

Syntax

```
#INCLUDE ADWPDIO.INC
PWM_OUT(module,channel,level)
```

Parameters

module	Specified module address	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

To be used for the modules

Pro-PWM-4, Pro-PWM-4-I

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  PWM_ENABLE(1,3)           'Enable counters of the PWM outputs
                            '1 and 2
  PWM_SET(1,1,0,2500,2500) 'Set PWM signal for output 1
  PWM_OUT(1,2,1)            'Set PWM output 2 to the locical value
                            '"1".
```

PWM_SET

The instruction **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 ADWPDIO.INC
PWM_SET(module,channel,prescale,low,high)
```

Parameters

module	Specified module address	LONG
channel	Number of the PWM output channel (1…4)	LONG
prescale	Exponent for the factor of the prescaler (see equation)	LONG
low	Value of the low-time (see equation)	LONG
high	Value of the high-time (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.

See also

PWM_ENABLE, PWM_OUT

To be used for the modules

Pro-PWM-4, Pro-PWM-4-I

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  PWM_ENABLE(1,3)           'Enable PWM outputs 1 and 2 for
                            'output
  PWM_SET(1,1,0,2500,2500)  'Set PWM signal for output 1
  PWM_OUT(1,2,1)            'Set PWM output 2 to the logical
                            'value "1"
```

SSI_MODE

The instruction **SSI_MODE** sets the modes of all SSI decoders on the specified module, either "single shot" (read out once) or "continous" (read out continously).

Syntax

```
#INCLUDE ADWPDIO.INC
SSI_MODE(module,pattern)
```

Parameters

module	Specified module address	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 continously.	LONG

Bit no.	31:2	1	0
SSI decoder	–	2	1

Notes

If you select the mode "continuous", reading the encoder is started immediately. The instruction **SSI_START** is not necessary for this.

Using the "continuous" mode, some encoder types occasionally return the wrong counter value 0 (zero) instead of the 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

To be used for the modules

Pro-CO4-D

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  SSI_SET_CLOCK(1,200)      'CLK (clock rate) = 50 kHz
  SSI_MODE(1,3)             'Set continuous-mode
                            '(for both encoders)
  SSI_SET_BITS(1,1,23)      'Amount of encoder bits=23 (encoder 1)
  SSI_SET_BITS(1,2,23)      'Amount of encoder bits=23 (encoder 2)

EVENT:
  PAR_1 = SSI_READ(1,1)     'Read out and display position value
                            '(encoder 1)
  PAR_2 = SSI_READ(1,2)     'Read out and display position value
                            '(encoder 2)
```

SSI_READ

The instruction **SSI_READ** returns the last saved counter value of a specified SSI counter on the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = SSI_READ(module,dcdr_no)
```

Parameters

module	Specified module address	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 the instruction **SSI_SET_BITS**, even if this does not correspond to the resolution of the encoder.
In this case the returned counter value depends on the encoder (see documentation of the manufacturer). Normally there are the following rules:

- If the encoder has a higher resolution, its exceeding least-significant bits are not used.
- If the encoder has a lower resolution as indicated, a 0 (zero) is read for each missing most-significant bit.

See also

SSI_MODE, SSI_SET_BITS, SSI_SET_CLOCK, SSI_START, SSI_STATUS

To be used for the modules

Pro-CO4-D

Example

```
#INCLUDE ADWPDIO.INC
DIM m, n, y AS LONG

INIT:
  SSI_SET_CLOCK(1,50)       'CLK (clock rate) = 200 kHz
  SSI_MODE(1,1)             'Set continuous-mode (encoder 1)
  SSI_SET_BITS(1,1,23)      'Amount of encoder bits=23 (encoder 1)

EVENT:
  PAR_1 = SSI_READ(1,1)     'Read out and display position
                            'value (encoder 1)
  REM If you have an encoder with Gray-code:
  m = 0                     'delete value of the last conversion
  y = 0                     ' -"-
  FOR n = 1 TO 32           'Check all 32 possible bits
    m = (SHIFT_RIGHT(PAR_1,(32 - n)) AND 1) XOR m
    y = (SHIFT_LEFT(m,(32 - n))) OR y
  NEXT n
  PAR_9 = y                 'The result of the Gray/binary
                            'conversion in PAR_9
```

SSI_SET_BITS

The instruction **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 ADWPDIO.INC
SSI_SET_BITS(module,dcdr_no,bit_no)
```

Parameters

module	Specified module address	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

To be used for the modules

Pro-CO4-D

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  SSI_SET_CLOCK(1,50)       'CLK (clock rate) = 200 kHz
  SSI_MODE(1,3)             'Set continuous-mode (for both
                            'encoders)
  SSI_SET_BITS(1,1,10)      '10 encoder bits for encoder 1
  SSI_SET_BITS(1,2,25)      '25 encoder bits for encoder 2

EVENT:
  PAR_1 = SSI_READ(1,1)     'Read out and display position value
                            '(encoder 1)
  PAR_2 = SSI_READ(1,2)     'Read out and display position value
                            '(encoder 2)
```

SSI_SET_CLOCK

The instruction **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 ADWPDIO.INC
SSI_SET_CLOCK(module,prescale)
```

Parameters

module	Specified module address	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.

Scale factors < 10 are automatically corrected to the value 10; from values > 255 only the least significant 8 bits are used as scale factor.

The possible clock frequency depends on the length of the cable, cable type, and the send and receive components of the encoder or decoder. Basically the following rule applies: The higher the clock frequency the shorter the cable length.

See also

SSI_MODE, SSI_READ, SSI_SET_BITS, SSI_START, SSI_STATUS

To be used for the modules

Pro-CO4-D

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  SSI_SET_CLOCK(1,10)       'CLK (clock rate) = 1 MHz
  SSI_MODE(1,3)             'Set continuous-mode
                            '(for both encoders)
  SSI_SET_BITS(1,1,10)      'Amount of encoder bits=10 (encoder 1)
  SSI_SET_BITS(1,2,25)      'Amount of encoder bits=25 (encoder 2)

EVENT:
  PAR_1 = SSI_READ(1,1)     'Read out and display position value
                            '(encoder 1)
  PAR_2 = SSI_READ(1,2)     'Read out and display position value
                            '(encoder 2)
```

SSI_START

The instruction **SSI_START** starts the reading of one or both SSI encoders on the specified module (only in mode "single shot").

Syntax

```
#INCLUDE ADWPDIO.INC
SSI_START(module,dcdr_no)
```

Parameters

module	Specified module address	LONG
dcdr_no	Bit pattern for selecting the SSI decoders which are to be started: Bit = 0: No function Bit = 1: Start reading of the SSI decoder	LONG

Bit no.	31:2	1	0
SSI decoder	–	2	1

Notes

In the continous mode this instruction has no function, because the encoder values are nevertheless read out continously.

An encoder value will be saved only when the amount of bits is read which is set by **SSI_SET_BITS**.

A complete encoder value is always transferred, even if the operation mode is changing meanwhile.

See also

SSI_MODE, SSI_READ, SSI_SET_BITS, SSI_SET_CLOCK, SSI_STATUS

To be used for the modules

Pro-CO4-D

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  SSI_SET_CLOCK(1,250)      'CLK (clock rate) = approx. 40 kHz
  SSI_MODE(1,0)             'Set single shot-mode
                            '(both counters)
  SSI_SET_BITS(1,1,23)      'Amount of encoder bits=23 (encoder 1)
  SSI_SET_BITS(1,2,23)      'Amount of encoder bits=23 (encoder 2)

EVENT:
  SSI_START(1,3)            'Read position value of encoders 1 & 2
  DO                        'for encoder 1:
  UNTIL (SSI_STATUS(1,1) = 0)
  REM If position value is read completely, then …
  PAR_1 = SSI_READ(1,1)     'read out and display position value
  DO                        'For encoder 2:
  UNTIL (SSI_STATUS(1,2) = 0)
  REM If position value is read completely, then …
  PAR_1 = SSI_READ(1,2)     'read out and display position value
```

SSI_STATUS

The instruction **SSI_STATUS** returns the current read-status on the speicified module for a specified decoder.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = SSI_STATUS(module,dcdr_no)
```

Parameters

module	Specified module address	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 the mode "continous" querying the status is not useful.

See also

SSI_MODE, SSI_READ, SSI_SET_BITS, SSI_SET_CLOCK, SSI_START

To be used for the modules

Pro-CO4-D

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  SSI_SET_CLOCK(1,250)      'CLK (clock rate) = approx. 40 kHz
  SSI_MODE(1,0)             'Set single shot-mode
                            '(both counters)
  SSI_SET_BITS(1,1,23)      'Amount of encoder bits=23 (encoder 1)
  SSI_SET_BITS(1,2,23)      'Amount of encoder bits=23 (encoder 2)

EVENT:
  SSI_START(1,3)            'Read position value of encoders 1 & 2
  DO                        'For encoder 1:
  UNTIL (SSI_STATUS(1,1) = 0)
  REM If position value is read completely, then …
  PAR_1 = SSI_READ(1,1)     'Read out and display position value
  DO                        'For encoder 2:
  UNTIL (SSI_STATUS(1,2) = 0)
  REM If position value is read completely, then …
  PAR_1 = SSI_READ(1,2)     'Read out and display position value
```

COMP_DIGIN_WORD

The instruction **COMP_DIGIN_WORD** returns the current status of the threshold value comparison for all channels of the specified module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = COMP_DIGIN_WORD(module)
```

Parameters

module	Specified module address	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

To be used for the modules

Pro-COMP16 Rev. A

Example

```
#INCLUDE ADWPDIO.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

The instruction **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 ADWPDIO.INC
ret_val = COMP_DIGIN_WORD_DIFF(module)
```

Parameters

module	Specified module address	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

To be used for the modules

Pro-COMP16 Rev. A

Example

```
#INCLUDE ADWPDIO.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

The instruction **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 ADWPDIO.INC
COMP_FIFO_READ(module,array1[],array2[])
```

Parameters

module	Specified module address	LONG
array1[]	Destination array for 1024 measurement values of the channel with the lower channel number.	ARRAY LONG
array2[]	Destination array for 1024 measurement values of the channel with the higher channel number.	ARRAY LONG

Notes

In the internal FIFO memory always the last 1024 measurement values of 2 channels are stored. You select the channel pair with **COMP_FIFO_SELECT**.

The instruction describes the array elements arrayx[1]...arrayx[1024] in both destination arrays. The destination arrays must be sufficiently dimensioned. The transferred measurement values range between 0…1024.

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-COMP16 Rev. A

Example

```
#INCLUDE ADWPDIO.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

The instruction **COMP_FIFO_SELECT** determines the channel pair whose data is stored in the internal FIFO memory of the module.

Syntax

```
#INCLUDE ADWPDIO.INC
COMP_FIFO_SELECT(module,ch_pair)
```

Parameters

module	Specified module address	LONG
ch_pair	Number (0…7) to determine a channel pair; the table below illustrates the assignment of the channels.	LONG

ch_pair	7	…	1	0
Channel no.	15, 16	…	3, 4	1, 2

Notes

Only one channel pair can be selected; it is not possible to determine more or fewer channel pairs.

The last 1024 measurement values of the two selected channels (= 2 × 1024 values) are written into the FIFO memory. The FIFO memory is read out with **COMP_FIFO_READ**.

See also

COMP_READ, COMP_RESET, COMP_SET, COMP_DIGIN_WORD, COMP_DIGIN_WORD_DIFF, COMP_FIFO_READ

To be used for the modules

Pro-COMP16 Rev. A

Example

```
#INCLUDE ADWPDIO.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

The instructions **COMP_READ** returns the current minimum or maximum measurement value of a specified channel on the module.

Syntax

```
#INCLUDE ADWPDIO.INC
ret_val = COMP_READ(module,channel,value)
```

Parameters

module	Specified module address	LONG
channel	Channel number (1…16)	LONG
value	New status for the selected output: 0: Current measurement value 1: Minimum measurement value 2: Maximum measurement value	LONG
ret_val	Measurement value (0…1023) of the selected channel in digits	LONG

Notes

A return value 0 digits corresponds to the minimum analog signal U_{min}, the value 1023 digits corresponds to the maximum analog signal U_{max}. To convert digits into Volt the following formula applies:

Conversion digits into Volt

$$\text{Volt} = \text{Digits} \cdot \frac{U_{max} - U_{min}}{1023} + U_{min}$$

This function will not delay or interrupt the measurement of the analog signal.

The minimum and maximum measurement values refer to the measurement values during the time period of the last reset with **COMP_RESET** (or to the moment of powering up the system).

See also

COMP_RESET, COMP_FIFO_SELECT, COMP_SET, COMP_DIGIN_WORD, COMP_DIGIN_WORD_DIFF, COMP_FIFO_READ

To be used for the modules

Pro-COMP16 Rev. A

Example

```
#INCLUDE ADWPDIO.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

The instruction COMP_RESET resets the measurement of the minimum and maximum values simultaneously for the selected channels.

Syntax

```
#INCLUDE ADWPDIO.INC
COMP_RESET(module,pattern)
```

Parameters

module	Specified module address	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

To be used for the modules

Pro-COMP16 Rev. A

Example

```
#INCLUDE ADWPDIO.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

The instruction **COMP_SET** determines the lower and upper threshold value for a specified channel.

Syntax

```
#INCLUDE ADWPDIO.INC
COMP_SET(module,channel,ValHigh,ValLow)
```

Parameters

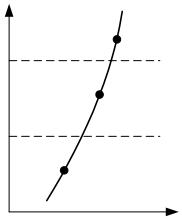
module	Specified module address	LONG
channel	Channel number(1…16)	LONG
ValHigh	Upper threshold value (1…1023) of the channel	LONG
ValLow	Lower threshold value (0…1022) of the channel	LONG

Notes

When an analog signal is evaluated you define a hysteresis by determining the threshold values.
Example: As long as there is a noisy analog signal near the switching point, the switching status (without hysteresis) changes more or less randomly. If the hysteresis threshold values are selected reasonably the switching status can be stabilized.

The upper threshold value must always be higher than the lower threshold value, otherwise the evaluation of the analog signals is not correct.

The analog signals (parallel on all channels) are measured with a fixed measurement rate (20 MHz 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 the instruction **COMP_DIGIN_WORD**.

In the same way, the difference value of 2 channels is compared with the threshold values and saved (1 bit for each of the channel pairs). The difference value is calculated as $value_1$ - $value_2$, $value_3$ - $value_4$, …, $value_{15}$ - $value_{16}$; the threshold values of the channel with the odd number are used.
The result of the comparison is read out with the instruction **COMP_DIGIN_WORD_DIFF**.

See also

COMP_READ, COMP_RESET, COMP_FIFO_SELECT, COMP_DIGIN_WORD, COMP_DIGIN_WORD_DIFF, COMP_FIFO_READ

To be used for the modules

Pro-COMP16 Rev. A

Example

```
#INCLUDE ADWPDIO.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

The instruction **RTC_SET** sets date and time on the real-time clock of the specified module. Invalid values are not accepted.

Syntax

```
#INCLUDE ADWPDIO.INC
RTC_SET(module,year,month,day,hour,minute,second)
```

Parameters

module	Specified module address	LONG
year	Year (0…63), corresponds to 2000…2063	LONG
month	Month (1…12)	LONG
day	Day (1…31); valid value ranges according to month and leap year.	LONG
hour	Hour (0…23)	LONG
minute	Minute (0…59)	LONG
second	Second (0…59)	LONG

Notes

The year refers to the time interval 2000…2063.

See also

RTC_GET

To be used for the modules

Pro-Storage Rev. A

Example

```
#INCLUDE ADWPDIO.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

The instruction RTC_GET returns date and time from the real-time clock of the specified module. Invalid values are not accepted.

Syntax

```
#INCLUDE ADWPDIO.INC
RTC_GET(module,year,month,day,hour,minute,second)
```

Parameters

module	Specified module address	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 which have been available for some time: RTC_GET_YEAR, RTC_GET_MONTH, RTC_GET_DAY, RTC_GET_HOUR, RTC_GET_MINUTE, RTC_GET_SECOND

See also

RTC_SET

To be used for the modules

Pro-Storage Rev. A

Example

```
#INCLUDE ADWPDIO.INC

INIT:
  REM Set real-time clock to 4.7.2003 9:17:30
  RTC_GET(1, 3,7,4,9,17,30)'on module 1
```

MEDIA_WR_BLK_L

The instruction **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 ADWPDIO.INC
ret_val = MEDIA_WR_BLK_L(module,f_info[],file,
          sec_idx,sec_cnt,array[],array_idx)
```

Parameters

module	Specified module address	LONG
f_info[]	Array, which contains the numbers of the start and end sectors of all files.	ARRAY LONG
file	Number (1…10) of the file	LONG
sec_idx	Index (1…m) of the first sector into which is written, referring to the start sector of the file. The max. value depends on the file size.	LONG
sec_cnt	Amount (1…12) of the data blocks to be transferred (= sectors) à 128 values.	LONG
array[]	Array whose data is transferred.	ARRAY 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.	31:07	07	06	05	04	03	02	01	00
	–	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.
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_WR_BLK_F, MEDIA_RD_BLK_L, MEDIA_RD_BLK_F, MEDIA_RD_ FILEINFO

To be used for the modules

Pro-Storage Rev. A

Example

```
REM ########################################################
REM Save sine table with 3600 points in a file
REM ########################################################
#INCLUDE ADWPDIO.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

The instruction **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 ADWPDIO.INC
ret_val = MEDIA_WR_BLK_F(module,f_info[],file,sec_
idx,
          sec_cnt,array[],array_idx)
```

Parameters

module	Specified module address	LONG
f_info[]	Array, which contains the numbers of the start and end sectors of all files.	ARRAY LONG
file	Number (1…10) of the file	LONG
sec_idx	Index (1…m) of the first sector into which is written, referring to the start sector of the file. The max. value depends on the file size.	LONG
sec_cnt	Amount (1…12) of the data blocks to be transferred (= sectors) à 128 values.	LONG
array[]	Array whose data is transferred.	ARRAY FLOAT
array_idx	Index (1…n) of the first array element to be transferred.	LONG
ret_val	Status of the glue-logic as bit pattern (see table)	LONG

Bit no.	31:07	07	06	05	04	03	02	01	00
	–	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.
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, MEDIA_RD_ FILEINFO

To be used for the modules

Pro-Storage Rev. A

MEDIA_RD_BLK_L

The instruction **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 ADWPDIO.INC
ret_val = MEDIA_RD_BLK_L(module,f_info[],file,
          sec_idx,sec_cnt,array[],array_idx)
```

Parameters

module	Specified module address	LONG
f_info[]	Array which contains the numbers of the start and end sectors of all files.	ARRAY LONG
file	Number (1…10) of the file	LONG
sec_idx	Index (1…m) of the first sector into which is written, referring to the start sector of the file. The max. value depends on the file size.	LONG
sec_cnt	Amount (1…12) of the data blocks to be read (=sectors) à 128 LONG values.	LONG
array[]	Destination array, into which the data is transferred.	ARRAY LONG
array_idx	Index (1…n) of the first array element into which is written.	LONG
ret_val	Status of the glue-logic as bit pattern (see below).	LONG

Bit No.	31:07	07	06	05	04	03	02	01	00
	–	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.
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_WR_BLK_F, MEDIA_RD_BLK_F, MEDIA_RD_ FILEINFO

To be used for the modules

Pro-Storage Rev. A

Example

```
REM ######################################################
REM read sine table with 3600 points from data file
REM ######################################################
#INCLUDE ADWPDIO.INC

#DEFINE pstom 1              'address of Pro-STORAGE module
#DEFINE f_info DATA_197      'array with file information
#DEFINE LUT DATA_1           'Look-Up table for sine
#DEFINE file 1               'File no. for reading the sine
#DEFINE nds 3600             'No. of points
REM the array length with the sine data must be a multiple
REM of 128 and greater or equal "nds":
#DEFINE lng 3712             'here: 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_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

The instruction **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 ADWPDIO.INC
ret_val = MEDIA_RD_BLK_F(module,f_info[],file,
          sec_idx,s_cnt,array[],array_idx)
```

Parameters

module	Specified module address	LONG
f_info[]	Array which contains the numbers of the start and end sectors of all files.	ARRAY LONG
file	Number (1…10) of the file	LONG
sec_idx	Index (1…m) of the first sector into which is written, referring to the start sector of the file. The max. value depends on the file size.	LONG
sec_cnt	Amount (1…12) of the data blocks to be read (= sectors) à 128 LONG values.	LONG
array[]	Destination array, into which the data is transferred.	ARRAY LONG
array_idx	Index (1…n) of the first array element into which is written.	LONG
ret_val	Status of the glue-logic as bit pattern (see below).	LONG

Bitnr.	31:07	07	06	05	04	03	02	01	00
	–	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.
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

MEDIA_WR_BLK_L, MEDIA_WR_BLK_F, MEDIA_RD_BLK_L, MEDIA_RD_ FILEINFO

To be used for the modules

Pro-Storage Rev. A

MEDIA_RD_ FILEINFO

The instruction **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 ADWPDIO.INC
ret_val = MEDIA_RD_FILEINFO(module,f_info[])
```

Parameters

module	Specified module address	LONG
f_info[]	Array which contains the numbers of the start and end sectors of all files.	ARRAY LONG
ret_val	Status of the glue-logic as bit pattern (see table)	LONG

Bit No.	31:06	05	04	03	02	01	00
	–	A_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_WR_BLK_F, MEDIA_RD_BLK_L, MEDIA_RD_BLK_F

To be used for the modules

Pro-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 ADWPDIO.INC

#DEFINE pstom 1              'address of Pro-STORAGE module
#DEFINE f_info DATA_197      'array with file information

DIM f_info[22] AS LONG AT DM_LOCAL 'array dimensioning
DIM n AS LONG                'local variables

INIT:
 PAR_52 = MEDIA_RD_FILEINFO(pstom,f_info)  'read file info
 REM check if medium is present and ready for read.
 REM Else -> EXIT
 IF ((PAR_52 AND 1001b) <> 1001b) THEN EXIT

 REM read all 10 files
 FOR n = 1 TO 10
  REM Calc. no. of first and last file sector
  PAR[n] = f_info[n*2 + 1]
  PAR[n+10] = f_info[n*2 + 2]
  REM file length > 1 sector?
  IF ((PAR[n+10] - PAR[n]) <> 0) THEN
   PAR[n+20] = PAR[n+10] - PAR[n] + 1 'No. of sectors
   PAR[n+30] = PAR[n+20] * 128 'No. of values
  ENDIF
 NEXT n

 EXIT
```

3.5 ADWPEXT.INC

The include file <ADWPEXT.inc> includes special system functions and procedures that are necessary to access the external *ADwin-Pro* modules. If you include this file with the *ADbasic* instruction

```
#INCLUDE ADWPEXT.INC
```

in your *ADbasic* program you can use all functions and procedures of this file.

On the following pages all functions of the file <ADWPEXT.inc> will be described more detailed. In addition an easy application example illustrates the functions.

TC_SELECT

The instruction **TC_SELECT** sets the thermocouple channels via multiplexer to the analog output of the module.

Syntax

```
#INCLUDE ADWPEXT.INC
TC_SELECT(module, channel)
```

Parameters

module	Specified module address	LONG
channel	Channel number: 1…4 (TC-4), 1…8 (TC-8) or 1…16 (TC-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)
- PT-x

You will find these tables in the *ADbasic* online help (or in the data sheet of the manufacturer: Analog Devices).

Can be used for the modules

Pro-TC-4, Pro-TC-8, Pro-TC-16

Pro-PT100-4, Pro-PT100-8

Example

It is presumed in the example that the analog output of the thermocouple module is connected with the analog input 1 of an A/D module with the module address 1.

```
#INCLUDE ADWPAD.INC
#INCLUDE ADWPEXT.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
```

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 ADWPEXT.INC
CAN_MSG[n] = value
```

or

```
ret_val = CAN_MSG[n]
```

Parameters

n	Number of an array element (1… 9)	LONG
value	Expression whose value (0...256) 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 the instruction TRANSMIT.

See also

EN_RECEIVE, EN_TRANSMIT, READ_MSG, TRANSMIT

To be used for the modules

Pro-CAN-1, Pro-CAN-2

Example

```
#INCLUDE ADWPEXT.INC

INIT:
  INIT_CAN(1,1)              'Initialize CAN controller
  EN_TRANSMIT(1,1,6,40,0)    'Enable message object 6 with 11 bit-
identifier 40 to send
  REM Send float number Pi (32-bit: 40490FDBh
  CAN_MSG[1] = 40h           'Byte 4 (MSB) &
  CAN_MSG[2] = 49h           'Byte 3 &…
  CAN_MSG[3] = 0Fh           'Byte 2 &…
  CAN_MSG[4] = 0DBh          'Byte 1
  CAN_MSG[9] = 4             'Length of the message in bytes

EVENT:
  TRANSMIT(1,1,6)            'Sends the message object 6
```

EN_RECEIVE

The instruction **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 ADWPEXT.INC
EN_RECEIVE(module,channel,objectno,id,extend)
```

Parameters

module	Specified module address	LONG
channel	Number (1, 2) of the CAN channel that determines the CAN controller	LONG
objectno	Number (1…15) of the message object in the CAN controller.	LONG
id	Identifier of the messages which are to be received in this message object (0…2^{11} or 0…2^{29}).	LONG
extend	Marker for the length of the identifier: 0: 11 bit identifier 1: 29 bit identifier	LONG

Notes:

A message object is only able to receive messages from the CAN bus, when it has been enabled before by **EN_RECEIVE**.

See also

CAN_MSG, EN_TRANSMIT, INIT_CAN, READ_MSG, TRANSMIT

To be used for the modules

Pro-CAN-1, Pro-CAN-2

Example

```
#INCLUDE ADWPEXT.INC

INIT:
  REM Initialization of the CAN controller 1 on the CAN module 1
  INIT_CAN(1,1)
  REM Enable message object 1 to receive CAN messages
  REM with the 11 bit-identifier 200
  EN_RECEIVE(1,1,1,200,0)
```

EN_TRANSMIT

The instruction **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 ADWPEXT.INC
EN_TRANSMIT(module,channel,objectno,id,extend)
```

Parameters

module	Specified module address	LONG
channel	Number (1, 2) of the CAN channel that specifies the CAN controller.	LONG
objectno	Number (1…14) of the message object in the CAN controller.	LONG
id	Identifier of the messages that are sent in this message object ($0…2^{11}$ oder $0…2^{29}$).	LONG
extend	Marker for the length of the identifier: 0: 11 bit identifier 1: 29 bit identifier	LONG

Notes

A message object can only transmit messages to the CAN bus when it has been enabled before by **EN_TRANSMIT**.

See also

CAN_MSG, EN_RECEIVE, INIT_CAN, READ_MSG, TRANSMIT

To be used for the modules

Pro-CAN-1, Pro-CAN-2

Example

```
#INCLUDE ADWPEXT.INC
INIT:
  REM Initialization of the CAN controller 1 on the CAN module 1
  INIT_CAN(1,1)
  REM Enable message object 6 for sending of CAN messages
  REM with the 11 bit-identifier 40
  EN_TRANSMIT(1,1,6,40,0)
```

GET_CAN_REG

The instruction **GET_CAN_REG** returns the contents of a specified register on a CAN controller on the specified module.

Syntax

```
#INCLUDE ADWPEXT.INC
ret_val = GET_CAN_REG(module,channel,regno)
```

Parameters

module	Specified module address	LONG
channel	Number (1, 2) of the CAN channel that determines the CAN controller	LONG
regno	Register number of the CAN controller (0…255)	LONG
ret_val	Contents of the CAN controller register	LONG

Notes

The register number that has to be indecated 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, SET_CAN_REG

To be used for the modules

Pro-CAN-1, Pro-CAN-2

Example

```
#INCLUDE ADWPEXT.INC
INIT:
  REM Initialization of the CAN controller 1 on the CAN module 1
  INIT_CAN(1,1)
  REM Read out the control register of CAN controller 1, module 1
  PAR_1 = GET_CAN_REG(1,1,0)
```

INIT_CAN

The instruction **INIT_CAN** initializes one of the CAN controllers on the specified module and sets it into an initial status.

Syntax

```
#INCLUDE ADWPEXT.INC
INIT_CAN(module,channel)
```

Parameters

module	Specified module address	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.

See also

EN_RECEIVE, EN_TRANSMIT, GET_CAN_REG, SET_CAN_BAUDRATE, SET_CAN_REG

To be used for the modules

Pro-CAN-1, Pro-CAN-2

Example

```
#INCLUDE ADWPEXT.INC
INIT:
  REM Initialization of the CAN controller 1 on the CAN module 1
  INIT_CAN(1,1)
```

READ_MSG

The instruction **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 ADWPEXT.INC
ret_val = READ_MSG(module,channel,msgno)
```

Parameters

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

Notes

The message object must be enabled before for receiving data with the instruction **EN_TRANSMIT**.

If the message object receives a new message, its data are read out and copied to the array CAN_MSG[]. In this case the return value corresponds to the identifier of the message received. The bits indicating that a message has been received are reset so that a new message can be received.

The instruction to be used for querying the message objects in the polling mode, because here a check is made first, if new messages have arrived. If this is not the case no error occurs in the program.

See also

CAN_MSG, EN_RECEIVE, EN_TRANSMIT, INIT_CAN, TRANSMIT

To be used for the modules

Pro-CAN-1, Pro-CAN-2

Example

```
REM Receives a message object and combines the individual bytes
REM if a new object with the right identifier has been received.
#INCLUDE ADWPEXT.INC
DIM n AS LONG
INIT:
  PAR_1 = 0
  INIT_CAN(1,1)              'Initialization of the CAN controller
  EN_RECEIVE(1,2,1,40,0)     'Enable message object 1 to receive
                             'messages with identifier 40

EVENT:
  REM Check object if messages have been received. If so the
  REM identifier is transferred to PAR_9 and the data bytes to
  REM the array CAN_MSG[].
  PAR_9 = READ_MSG(1,2,1)
  REM Have new messages with the identifier 40 arrived?
  IF (PAR_9 = 40) THEN
    PAR_1 = CAN_MSG[1]       'Read out high-byte auslesen
    FOR n = 2 TO 4           'Read out further 3 bytes ...
      REM ... and combine all to a 32-bit number.
      PAR_1 = SHIFT_LEFT(PAR_1,8) + CAN_MSG[n]
    NEXT n
  ENDIF
```

SET_CAN_BAUDRATE

The instruction **SET_CAN_BAUDRATE** sets the baud rate on one of the controllers on the specified module and returns the information if this was successful.

Syntax

```
#INCLUDE ADWPEXT.INC
ret_val = SET_CAN_BAUDRATE(module,channel,rate)
```

Parameters

module	Specified module address	LONG
channel	Number (1, 2) of the CAN channel, that determines the CAN controller.	LONG
rate	Baud rate of the CAN controller in bits per second.	LONG
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 (1 MBit/s).

See also

GET_CAN_REG, INIT_CAN, SET_CAN_REG

To be used for the modules

Pro-CAN-1, Pro-CAN-2

Example

```
#INCLUDE ADWPEXT.INC
DIM status AS LONG
INIT:
  INIT_CAN(1,1)              'Initialization of the CAN controller
  status = SET_CAN_BAUDRATE(1,1,125000)'Set Baud rate 125 kBit/s
```

Available baud rates

Available baud rates [Bit/s]							
1000000.0000	888888.8889	800000.0000	727272.7273	666666.6667	615384.6154	571428.5714	533333.3333
500000.0000	470588.2353	444444.4444	421052.6316	400000.0000	380952.3810	363636.3636	347826.0870
333333.3333	320000.0000	307692.3077	296296.2963	285714.2857	266666.6667	250000.0000	242424.2424
235294.1176	222222.2222	210526.3158	205128.2051	200000.0000	190476.1905	181818.1818	177777.7778
173913.0435	166666.6667	160000.0000	156862.7451	153846.1538	148148.1481	145454.5455	142857.1429
140350.8772	133333.3333	126984.1270	125000.0000	123076.9231	121212.1212	117647.0588	115942.0290
114285.7143	111111.1111	106666.6667	105263.1579	103896.1039	102564.1026	100000.0000	98765.4321
95238.0952	94117.6471	90909.0909	88888.8889	87912.0879	86956.5217	84210.5263	83333.3333
81632.6531	80808.0808	80000.0000	78431.3725	76923.0769	76190.4762	74074.0741	72727.2727
71428.5714	70175.4386	69565.2174	68376.0684	67226.8908	66666.6667	66115.7025	64000.0000
63492.0635	62500.0000	61538.4615	60606.0606	60150.3759	59259.2593	58823.5294	57971.0145
57142.8571	55944.0559	55555.5556	54421.7687	53333.3333	52631.5789	52287.5817	51948.0519
51282.0513	50000.0000	49689.4410	49382.7160	48484.8485	47619.0476	47337.2781	47058.8235
46783.6257	45714.2857	45454.5455	44444.4444	43956.0440	43478.2609	42780.7487	42328.0423
42105.2632	41666.6667	41025.6410	40816.3265	40404.0404	40000.0000	39215.6863	38647.3430
38461.5385	38277.5120	38095.2381	37037.0370	36363.6364	36199.0950	35714.2857	35555.5556
35087.7193	34782.6087	34632.0346	34482.7586	34188.0342	33613.4454	33333.3333	33057.8512
32921.8107	32388.6640	32258.0645	32000.0000	31746.0317	31620.5534	31372.5490	31250.0000
30769.2308	30651.3410	30303.0303	30075.1880	29629.6296	29411.7647	29304.0293	29090.9091
28985.5072	28673.8351	28571.4286	28070.1754	27972.0280	27777.7778	27681.6609	27586.2069
27210.8844	27027.0270	26936.0269	26755.8528	26666.6667	26315.7895	26143.7908	25974.0260
25806.4516	25641.0256	25396.8254	25078.3699	25000.0000	24844.7205	24767.8019	24691.3580
24615.3846	24390.2439	24242.4242	24024.0240	23809.5238	23668.6391	23529.4118	23460.4106
23391.8129	23255.8140	23188.4058	22988.5057	22857.1429	22792.0228	22727.2727	22408.9636
22222.2222	22160.6648	22038.5675	21978.0220	21739.1304	21680.2168	21621.6216	21505.3763
21390.3743	21333.3333	21276.5957	21220.1592	21164.0212	21052.6316	20833.3333	20779.2208
20671.8346	20512.8205	20460.3581	20408.1633	20202.0202	20050.1253	20000.0000	19851.1166
19753.0864	19704.4335	19656.0197	19607.8431	19512.1951	19323.6715	19230.7692	19138.7560
19047.6190	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

Available baud rates [Bit/s]							
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	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

The instruction **SET_CAN_REG** writes a value in a register of the selected CAN controller on the specified module.

Syntax

```
#INCLUDE ADWPEXT.INC
SET_CAN_REG(module,channel,regno,value)
```

Parameters

module	Specified module address	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

To be used for the modules

Pro-CAN-1, Pro-CAN-2

Example

```
#INCLUDE ADWPEXT.INC
INIT:
  INIT_CAN(1,1)              'Initialization of the CAN controller
  SET_CAN_REG(1,1,0,1)       'Set control register to the value 1
```

TRANSMIT

The instruction **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 ADWPEXT.INC
TRANSMIT(module,channel,msgno)
```

Parameters

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

Notes

This instruction can only be executed when the corresponding message object has been configured before to send with the **EN_TRANSMIT**.

The message data must be entered in CAN_MSG [], before executing the instruction **TRANSMIT**.

See also

CAN_MSG, EN_RECEIVE, EN_TRANSMIT, READ_MSG

To be used for the modules

Pro-CAN-1, Pro-CAN-2

Example

```
REM Send a message object (the number Pi as 32-bit FLOAT number)
#INCLUDE ADWPEXT.INC
INIT:
  INIT_CAN(1,1)              'Initializing the CAN controller
  REM Initialize message object 6 to send with identifier 40
  EN_TRANSMIT(1,1,6,40,0)
  REM Send float number Pi (32-bit): 40490FDBh
  CAN_MSG[1] = 40h           'Byte 4 (MSB) &
  CAN_MSG[2] = 49h           'Byte 3 &…
  CAN_MSG[3] = 0Fh           'Byte 2 &…
  CAN_MSG[4] = 0DBh          'Byte 1
  CAN_MSG[9] = 4             'Length of the message in bytes

EVENT:
  TRANSMIT(1,1,6,40,0)       'Sends the message object
```

CHANGED_DATA

The function **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 ADWPEXT.INC
ret_val = CHANGED_DATA(module,outsize)
```

Parameters

module	Specified module address	LONG
outsize	Size in bytes of the output area to check	LONG
ret_val	Flag for data changes in the output area: 0 Data have not been changed ≠0 New data available	LONG

Notes

This function is used to save computing time, by only reading out the data when they are changed.

The flag is reset when the right to access the output area changes from the application to the fieldbus. It doesn't matter if the function **CHANGED_DATA** is called or not.

The function can only be used when you have released it during initialization (see **INIT_SLAVE**).

To be used for the modules

Inter-SL, Profi-SL

See also

CHECK_ACCESS, GET_PRO_BYTE, GET_READ_BUFFER, INIT_SLAVE, REQUEST_ACCESS, REQUEST_RELEASE_ACCESS, SET_PRO_BYTE, SET_WRITE_BUFFER

Example

```
#INCLUDE ADWPEXT.INC

EVENT:
  REQUEST_ACCESS(1,2)       'Request access to the DP-RAM
                            '(Output area)
  PAR_1 = CHECK_ACCESS(1)   'Get access rights status
  IF (PAR_1 = 2) THEN       'If ADwin has access rights ...
    IF (CHANGED_DATA(1,10)<>0) THEN 'and if there are new data ...
      PAR_2 = GET_PRO_BYTE(1,10) 'a byte is read.
    ENDIF
  ENDIF
  REQUEST_RELEASE_ACCESS(1,2) 'Release access to the DPM-RAM
```

CHECK_ACCESS

The function CHECK_ACCESS returns to which area of the DP-RAM the application has access rights.

Syntax

```
#INCLUDE ADWPEXT.INC
ret_val = CHECK_ACCESS(module)
```

Parameters

module	Specified module address	LONG
ret_val	Flag for access rights to a data area 0: Control area 1: Output area 2: Input area	LONG

Notes

If the application has no access right for a data area of the DP-RAM, you cannot access this area. This is necessary because otherwise the data will become inconsistent and the system may not work appropriately.

To be used for the modules

Inter-SL, Profi-SL

See also

CHANGED_DATA, GET_PRO_BYTE, GET_READ_BUFFER, INIT_SLAVE, REQUEST_ACCESS, REQUEST_RELEASE_ACCESS, SET_PRO_BYTE, SET_WRITE_BUFFER

Example

```
#INCLUDE ADWPEXT.INC

EVENT:
  REQUEST_ACCESS(1,1)       'Request access to the DP-RAM
                            '(Control register).
  PAR_1 = CHECK_ACCESS(1)   'Check, if ADwin has access rights
  IF (PAR_1 = 1) THEN       'If there is access right, ...
    PAR_2 = GET_PRO_BYTE(1,7F6h) 'a byte is read
  ENDIF
  REQUEST_RELEASE_ACCESS(1,1) 'Release access to the DP-RAM.
```

GET_PRO_BYTE

The function **GET_PRO_BYTE** returns a byte of a specified memory address of the DP-RAM of the fieldbus module.

Syntax

```
#INCLUDE ADWPEXT.INC
ret_val = GET_PRO_BYTE(module,byteno)
```

Parameter

module	Specified module address	LONG
byteno	Memory address in the DP-RAM	LONG
ret_val	Contents of the memory address of the DP-RAM (0…256)	LONG

Notes

The function accesses each memory location, regardless 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.

To be used for the modules

Inter-SL, Profi-SL

See also

CHANGED_DATA, CHECK_ACCESS, GET_READ_BUFFER, INIT_SLAVE, REQUEST_ACCESS, REQUEST_RELEASE_ACCESS, SET_PRO_BYTE, SET_WRITE_BUFFER

Example

```
#INCLUDE ADWPEXT.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

The instruction **GET_READ_BUFFER** copies a defined data block from the memory area of the DP-RAM into the specified destination array.

Syntax

```
#INCLUDE ADWPEXT.INC
GET_READ_BUFFER(module,array[],start,count)
```

Parameter

module	Specified module address	LONG
array[]	Name of the destination array	ARRAY LONG
start	First element in the DP-RAM of the data block, which is to copy	LONG
count	Amount of data bytes to copy from the DP-RAM	LONG

Notes

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

The destination array into which the data are transferred, must already be declared, at least with so many array elements as data bytes are copied.

To be used for the modules

Inter-SL, Profi-SL

See also

CHANGED_DATA, CHECK_ACCESS, GET_PRO_BYTE, INIT_SLAVE, REQUEST_ACCESS, REQUEST_RELEASE_ACCESS, SET_PRO_BYTE, SET_WRITE_BUFFER

Example

```
#INCLUDE ADWPEXT.INC
DIM DATA_1[100] AS LONG

EVENT:
  REQUEST_ACCESS(1,2)       'Request access to the DP-RAM
                            '(output area)
  PAR_1 = CHECK_ACCESS(1)   'Read access right status
  IF (PAR_1 = 2) THEN       'If ADwin has access right...
    IF (CHANGED_DATA(1,10) <> 0) THEN 'and new data are available
      GET_READ_BUFFER(1,DATA_1,0,10) '10 bytes are read out.
    ENDIF
  ENDIF
  REQUEST_RELEASE_ACCESS(1,2) 'Release access to the DP-RAM.
```

INIT_SLAVE

The instruction **INIT_SLAVE** initializes the fieldbus slave and can only be used after power up.

Syntax

```
#INCLUDE ADWPEXT.INC
ret_val = INIT_SLAVE(module, IO_in, par_in, IO_out,
par_out, in_hdl, out_hdl, intr)
```

Parameters

module Specified module address LONG

IO_in Size of the input data area in bytes for cyclic data LONG

par_in Length of the input data area in bytes for acyclid data LONG

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

par_out Length of the output data area in bytes for acyclic data LONG

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

Bit no.	31:2	1	0
Bit = 0	–	Disable flag for **CHANGED_DATA**.	Clear input area as soonas 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.

intr Bit pattern for the handling of the interrupt LONG

Bit no.	31:2	1	0
	–	Release interrupt when the fieldbus is on-line:	Release interrupt when the flag for **CHANGED_DATA** is enabled:
Bit = 0	–	No	No
Bit = 1	–	Yes	Yes

ret_val — Bit pattern as status of the initialization: LONG
Bit = 0: no error
Bit = 1: error occurred (meaning see below)

Bit 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.

After a successful initialization the slave is ready to exchange data and parameters can be read out from the DP-RAM. The application is only allowed to write information into the DP-RAM when it has access rights!

If incorrect data have been exchanged during the first initialization, the system must be turned off. Only after a new power up can the module be reinitialized.

With this instruction a sequence is processed, which takes approx. 2-3 seconds. If the instruction is called in a high-priority process (that cannot be interrupted), there will be no communication between computer and *ADwin* system. Therefore we recommend to initialize in a low-priority process or in a process section with low priority (e.g. **LOWINIT:**) .

To be used for the modules

Inter-SL, Profi-SL

See also

CHANGED_DATA, CHECK_ACCESS, GET_PRO_BYTE, GET_READ_BUFFER, REQUEST_ACCESS, REQUEST_RELEASE_ACCESS, SET_PRO_BYTE, SET_WRITE_BUFFER

Example

```
#INCLUDE ADWPEXT.INC

INIT:
  PAR_1 = INIT_SLAVE(1,10,0,10,0,2,0,0)
  REM The slave is initialized::
  REM Only cyclic data (10 IO_in / 10 IO_out),
  REM CHANGE_DATA function ON, no interrupts,
  REM Outputs are cleared, when the bus is OFF-line.
```

REQUEST_ACCESS

The instruction **REQUEST_ACCESS** requests access to the DP-RAM of the slave.

Syntax

```
#INCLUDE ADWPEXT.INC
REQUEST_ACCESS(module,area)
```

Parameters

module	Specified module address	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.

To be used for the modules

Inter-SL, Profi-SL

See also

CHANGED_DATA, CHECK_ACCESS, GET_PRO_BYTE, GET_READ_BUFFER, INIT_SLAVE, REQUEST_RELEASE_ACCESS, SET_PRO_BYTE, SET_WRITE_BUFFER

Example

```
#INCLUDE ADWPEXT.INC

EVENT:
  REQUEST_ACCESS(1,1)       'Request access to the DP-RAM
                            '(Control register).
  PAR_1 = CHECK_ACCESS(1)   'Get access rights status
  IF (PAR_1 = 1) THEN       'If ADwin has access right...
    PAR_2 = GET_PRO_BYTE(1,7F6h) 'a byte is read.
  ENDIF
  REQUEST_RELEASE_ACCESS(1,1) 'Return access to the DP-RAM.
```

REQUEST_RELEASE_ACCESS

The instruction **REQUEST_RELEASE_ACCESS** requests to return the access right for the DP-RAM of the slave to the fieldbus.

Syntax

```
#INCLUDE ADWPEXT.INC
REQUEST_RELEASE_ACCESS(module,area)
```

Parameters

module	Specified module address	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.

To be used for the modules

Inter-SL, Profi-SL

See also

CHANGED_DATA, CHECK_ACCESS, GET_PRO_BYTE, GET_READ_BUFFER, INIT_SLAVE, REQUEST_ACCESS, SET_PRO_BYTE, SET_WRITE_BUFFER

Example

```
#INCLUDE ADWPEXT.INC

EVENT:
  REQUEST_ACCESS(1,1)       'Request access to the DP-RAM
                            '(Control register).
  PAR_1 = CHECK_ACCESS(1)   'Get access rights status
  IF (PAR_1 = 1) THEN       'If ADwin has access right...
    PAR_2 = GET_PRO_BYTE(1,7F6h) 'a byte is read.
  ENDIF
  REQUEST_RELEASE_ACCESS(1,1) 'Release access to the DP-RAM.
```

SET_PRO_BYTE

The instruction **SET_PRO_BYTE** sets a byte in the DP-RAM of the fieldbus slave.

Syntax

```
#INCLUDE ADWPEXT.INC
SET_PRO_BYTE(module,byteno,value)
```

Parameters

module	Specified module address	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.

To be used for the modules

Inter-SL, Profi-SL

See also

CHANGED_DATA, CHECK_ACCESS, GET_PRO_BYTE, GET_READ_BUFFER, INIT_SLAVE, REQUEST_ACCESS, REQUEST_RELEASE_ACCESS, SET_WRITE_BUFFER

Example

```
#INCLUDE ADWPEXT.INC
INIT:
SET_PRO_BYTE(1,0h,2)'The byte number 0h is set to the value 2
(cyclic input data)
```

SET_WRITE_BUFFER

The instruction **SET_WRITE_BUFFER** copies the data from an array into a specified memory area of the DP-RAM.

Syntax

```
#INCLUDE ADWPEXT.INC
SET_WRITE_BUFFER(module,array[],start,count)
```

Parameters

module	Specified module address	LONG
array[]	Name of the source array	ARRAY LONG
start	First memory byte in the DP-RAM into which data are written	LONG
count	Amount of memory bytes in the DP-RAM into which data are written	LONG

Notes

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

The source array that provides the data, must already be declared, at least with so many array elements as data bytes are copied.

To be used for the modules

Inter-SL, Profi-SL

See also

CHANGED_DATA, CHECK_ACCESS, GET_PRO_BYTE, GET_READ_BUFFER, INIT_SLAVE, REQUEST_ACCESS, REQUEST_RELEASE_ACCESS, SET_PRO_BYTE

Example

```
#INCLUDE ADWPEXT.INC

DIM DATA_1[100] AS LONG

EVENT:
  REQUEST_ACCESS(1,4)       'request access to the DP-RAM
                            '(input area)
  PAR_1 = CHECK_ACCESS(1)   'Get access rights status
  IF (PAR_1 = 2) THEN       'If ADwin has access right...
    IF (CHANGED_DATA(1,10) <> 0) THEN 'and if there are new data
      GET_READ_BUFFER(1,DATA_1,0,10) '10 bytes are transferred.
    ENDIF
  ENDIF
  REQUEST_RELEASE_ACCESS(1,4) 'Release access to the DP-RAM.
```

CHECK_SHIFT_REG

The instruction **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 ADWPEXT.INC
ret_val = CHECK_SHIFT_REG(module,register)
```

Parameters

module	Specified module address	LONG
channel	number of the channel that is to be read out (1, 2 or 1…4)	LONG
ret_val	Sending status: 0: Data has been sent (= no more data in the send-FIFO). 1: Not yet all data sent (= the send-FIFO still contains data).	LONG

Notes

With the return value 0 both the send FIFO and the output shift register are empty. With the return value 1 there is at least one bit to be sent.

We recommend to use this instruction only after you have more experience about how the controller operates (data-sheet of the manufacturer Texas Instruments). For more common applications more comfortable instructions are availabe in the include file.

See also

CHECK_SHIFT_REG, READ_FIFO, RS_INIT, RS_RESET, RS485_SEND, SET_RS, WRITE_FIFO

To be used for the modules

RS232-2, RS232-4

RS485-2, RS485-4

Example

```
#INCLUDE ADWPEXT.INC

EVENT:
  …
  PAR_1 = CHECK_SHIFT_REG(1,1)'Check if channel 1 still
                              'has data to send
  …
```

GET_RS

The instruction **GET_RS** reads out the controller register on the specified module.

Syntax

```
#INCLUDE ADWPEXT.INC
ret_val = GET_RS(module,register)
```

Parameters

module	Specified module address	LONG
register	Address of the controller register to read	LONG
ret_val	Contents of the controller register	LONG

Notes

We recommend to use this instruction only after you have more experience about how the controller operates (data-sheet of the manufacturer Texas Instruments). For more common applications more comfortable instructions are availabe in the include file.

See also

CHECK_SHIFT_REG, READ_FIFO, RS_INIT, RS_RESET, RS485_SEND, SET_RS, WRITE_FIFO

To be used for the modules

RS232-2, RS232-4

RS485-2, RS485-4

Example

-/-

READ_FIFO

The instruction **READ_FIFO** reads a value from the input FIFO of a specified channel on the specified module.

Syntax

```
#INCLUDE ADWPEXT.INC
result = READ_FIFO(module,channel)
```

Parameters

module	Specified module address	LONG
channel	number of the channel that is to be read out (1, 2 or 1…4)	LONG
ret_val	Contents of the input FIFO: -1: FIFO is empty ≥0:Transferred value	LONG

Notes

-/-

See also

CHECK_SHIFT_REG, GET_RS, RS_INIT, RS_RESET, RS485_SEND, SET_RS, WRITE_FIFO

To be used for the modules

RS232-2, RS232-4

RS485-2, RS485-4

Example

```
#INCLUDE ADWPEXT.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.

EVENT:
  PAR_1 = READ_FIFO(1,1)   'Get a value from the FIFO. If
                           'the FIFO is empty, -1 is returned.
```

RS_INIT

The instruction **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 ADWPEXT.INC
RS_INIT(module,channel,baud,parity,bits,stop,
        handshake)
```

Parameters

module	Specified module address	LONG
channel	Channel, which is to be initialized (1, 2 or 1…4)	LONG
baud	Transfer rate in 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) 2: Software handshake (Xon/Xoff) 3: RS485	LONG

Notes

This instruction is necessary before working first with the selected channel, in order to set the interface parameters. They must be identical to the remote station, in order to verify a correct transfer.

The initialization is necessary after you have executed a hardware reset with the instruction **RS_RESET**.

See also

CHECK_SHIFT_REG, GET_RS, READ_FIFO, RS_RESET, RS485_SEND, SET_RS, WRITE_FIFO

To be used for the modules

RS232-2, RS232-4

RS485-2, RS485-4

Example

```
#INCLUDE ADWPEXT.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.
```

RS_RESET

The instruction **RS_RESET** executes a hardware reset on the specified module and deletes the settings for all channels.

Syntax

```
#INCLUDE ADWPEXT.INC
RS_RESET(module)
```

Parameters

module	Specified module address	LONG

Notes

The instruction sends a reset impulse to the input of the controller TL16C754. In the data-sheet of the controller 16C754 from Texas Instruments it is described, to which values the registers have been set after the hardware reset.

After a hardware reset an initialization with **RS_INIT** must follow, in order to initialize the controller and to set the interface parameters.

The instruction **RS_INIT** sets the same registers as a hardware reset does. Nevertheless, **RS_RESET** should be used for the case the controller has crashed.

See also

CHECK_SHIFT_REG, GET_RS, READ_FIFO, RS_INIT, RS485_SEND, SET_RS, WRITE_FIFO

To be used for the modules

RS232-2, RS232-4

RS485-2, RS485-4

Example

```
#INCLUDE ADWPEXT.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.
```

RS485_SEND

The instruction **RS485_SEND** determines the transfer direction for a specified channel on the specified module.

Syntax

```
#INCLUDE ADWPEXT.INC
RS485_SEND(module,channel,dir)
```

Parameters

module	Specified module address	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

To be used for the modules

RS485-2, RS485-4

Example

-/-

SET_RS

The instruction **SET_RS** writes a value into a specified register on the specified module.

Syntax

```
#INCLUDE ADWPEXT.INC
SET_RS(module,register,value)
```

Parameters

module	Specified module address	LONG
register	Number of the register, into which data are written	LONG
value	Value to be written into the register	LONG

Notes

We recommend to use this instruction only after you have more experience about how the controller operates (data-sheet of the manufacturer: TL16C754 from Texas Instruments). For more common applications more comfortable instructions are availabe in the include file.

See also

CHECK_SHIFT_REG, GET_RS, READ_FIFO, RS_INIT, RS_RESET, RS485_SEND, WRITE_FIFO

To be used for the modules

RS232-2, RS232-4

RS485-2, RS485-4

Example

-/-

WRITE_FIFO

The instruction **WRITE_FIFO** writes a value into the send-FIFO of a specified channel on the specified module.

Syntax

```
#INCLUDE ADWPEXT.INC
ret_val = WRITE_FIFO(module,channel,value)
```

Parameters

module	Specified module address	LONG
channel	Channel number to whose send-FIFO data are transferred (1, 2 or 1…4)	LONG
value	Value to be written into the send-FIFO	LONG
ret_val	Status message: 0: Data are transferred successfully. 1: Data were not transferred, send-FIFO is full.	LONG

Notes

The instruction checks first if there is at least one memory space in the send-FIFO. If this is so, the transferred value is written into the FIFO (return value 0); otherwise a 1 is returned, indicating that the FIFO is full and writing is not possible.

See also

CHECK_SHIFT_REG, GET_RS, READ_FIFO, RS_INIT, RS_RESET, RS485_SEND, SET_RS

Can be used for the modules

RS232-2, RS232-4

RS485-2, RS485-4

Example

```
#INCLUDE ADWPEXT.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.

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).
```

4 Program Examples

For reasonable use of the examples an external controlled system should be connected to the Pro system.

4.1 Online Evaluation of Measurement Data

The program <PRO_DMO1.BAS> searches for the maximum and minimum value out of 1000 measurements of ADC1 and writes the result to the variables PAR_1 und PAR_2.

The program assumes that a 16-bit A/D module is used and that the module address is set to 1.

```
#INCLUDE adwpad.inc     'Include-file for Pro A/D modules
#DEFINE limit 65535     'max. 16 bit ADC-value
DIM i1, iw, max, min AS INTEGER

INIT:
  i1 = 1                'reset sample counter
  max = 0               'initial maximum value
  min = limit           'initial minimum value
  PAR_10 = 0            'init End-Flag
  GLOBALDELAY = 40000   'cycle-time of 1ms (T9)

EVENT:
  iw = ADC16(1,1)       'get sample
  IF (iw > max) THEN max = iw 'new maximum sample?
  IF (iw < min) THEN min = iw 'new minimum sample?
  INC i1                'increment index
  IF (i1 > 1000) THEN   '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
'   ACTIVATE_PC         'only for use with TestPoint
    PAR_10 = 1          'set End-Flag
  ENDIF
```

4.2 Digital Proportional Controller

The program <PRO_DMO2.BAS> is a digital proportional controller. The set-point is specified by PAR_1, the gain by PAR_2.

The program assumes that a 16-bit A/D module is used and that the module address is set to 1. The address on the D/A module should also be set to 1.

```
#INCLUDE adwpad.inc     'Include file for Pro A/D modules
#INCLUDE adwpda.inc     'Include file for Pro D/A modules

#DEFINE offset 32768    '0V for 16 bit ADC/DAC-systems

REM cd: control deviation; av: actuating value
DIM cd, av AS INTEGER

INIT:
  PAR_1 = offset        'initial setpoint
  PAR_2 = 10            'initial gain
  GLOBALDELAY = 40000   'cycle-time of 1ms (T9)

EVENT:
  cd = PAR_1 - ADC16(1, 1)'compute control deviation (cd)
  av = cd * PAR_2 + offset'compute actuating value (av)
  DAC(1, 1, av)         'output actuating value on DAC-#1
```

4.3 Data Exchange with DATA arrays

The program <PRO_DMO3.BAS> measures the analog input 1 of the A/D module with address 1 and sets an end flag after 1000 measurements to indicate that the computer can now get the measurement data. The data are transferred by using the array DATA_1.

```
#INCLUDE adwpad.inc     'Include-file for Pro A/D modules

DIM DATA_1[1000] AS INTEGER
DIM index AS INTEGER

INIT:
  PAR_10 = 0
  index = 0             'reset array pointer
  GLOBALDELAY = 40000   'cycle-time of 1ms (T9)

EVENT:
  index = index + 1     'increment array pointer
  IF (index > 1000) THEN'1000 samples done?
'   ACTIVATE_PC         'set ACTIVATE_PC flag (only necessary
                        'for TestPoint)
    PAR_10 = 1          'set End-Flag
    END                 'terminate process
  ENDIF
  DATA_1[index] = ADC16(1, 1) 'acquire sample and save in array
```

4.4 Digital PID Controller

The program <PRO_DMO6.BAS> is a digital PID controller.

Before starting the PID controller the global variables must be set to the controller's values.

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 25ns
Information from the program	
PAR_5	array index (of DATA_1) of control deviation
PAR_9	mean value of control deviation
PAR_10	Flag: All samples are done
DATA_1[]	Array holding all control deviations

***ADbasic* Program:**

Both the addresses of the analog input and analog output module are set to the address 1 in the example.

Please note that you will get a time saving effect, when calculation and output of the actuating value is executed during the necessary waiting period during reading the control deviation (after SET_MUX and START_CONV).

The consequence is that the output actuating value is calculated from the control deviation of the previous process call.

```
#INCLUDE adwpad.inc     'Include file for Pro A/D modules
#INCLUDE adwpda.inc     'Include file for Pro D/A modules

#DEFINE offset 32768    '0V output

DIM DATA_1[4000] AS LONG
DIM av, cd, cdo, sum AS LONG
DIM diff AS FLOAT

INIT:
  sum = 0               'initial value of integral part
  cd = ADC(1)           'initial value of control deviation
                        '(cd) & MUX to Ch-#1
  PAR_5 = 1             'set array index
  IF (FPAR_3 < 1E3) THEN FPAR_3 = 1E3'check min. of integration
                        'time
  IF (PAR_6 < 4E4) THEN PAR_6 = 4E4'allow cycle-times >= 1ms
  GLOBALDELAY = PAR_6   'set cycle-time

EVENT:
  REM compute actuating value
  av = FPAR_2 * (cd + sum / FPAR_3 + diff * FPAR_4)
  START_CONV(1)         'start conversion ADC-#1
  REM while conversion is running …
  DAC(1, av + offset)   'output actuating value at DAC-#1
  cdo = cd              'keep control deviation in mind
  WAIT_EOC(1)           'wait until end-of-conversion of ADC
  cd = PAR_1 - READADC(1)'compute control deviation
  FPAR_9 = FPAR_9 * 0.99 + cd * 0.01'mean value of control
                        'deviation
  sum = sum + cd        'calculate integral
  IF (sum > 2E6) THEN sum = 2E6 'positive limit of integral
  IF (sum < -2E6) THEN sum = -2E6 'negative limit of integral
  diff = (cd - cdo)     'calculate deviation difference
  DATA_1[PAR_5] = cd    'write control deviation in a buffer
  INC PAR_5             'increment buffer index
  IF (PAR_5 >= 4000) THEN'4000 samples done?
'  ACTIVATE_PC          'only for use with TestPoint
   PAR_10 = 1           'set End-flag
   PAR_5 = 1            'reset array index
  ENDIF

FINISH:
  DAC(1,offset)         'analog output #1 to 0V
```

Annex

A-1 Alphabetic Instruction List

A-2 Instruction List sorted by Module Types

AIn-32/12 Rev. A

A: ADC · 18
C: CHECKLED · 4
R: READADC · 38
S: SETLED · 8
SET_MUX · 46
SE_DIFF · 44
START_CONV · 62
SYNCALL · 11
SYNCENABLE · 13
SYNCSTAT · 15

AIn-32/12 Rev. B

A: ADC · 18
C: CHECKLED · 4
R: READADC · 38
READADC_SCONV · 39
S: SETLED · 8
SET_MUX · 46
SE_DIFF · 44
START_CONV · 62
SYNCALL · 11
SYNCENABLE · 13
SYNCSTAT · 15
W: WAIT_EOC · 66

AIn-32/14 Rev. A

A: ADC · 18
C: CHECKLED · 4
R: READADC · 38
READADC_SCONV · 39
S: SEQ_MODE · 48
SEQ_READ · 50
SEQ_READ32 · 56
SEQ_READ_ONE · 51
SEQ_READ_PACKED · 54
SEQ_READ_TWO · 52
SEQ_SELECT · 57
SEQ_SET_DELAY · 58
SEQ_STATUS · 60
SETLED · 8
SET_MUX · 46
SE_DIFF · 44
START_CONV · 62
SYNCALL · 11
SYNCENABLE · 13
SYNCSTAT · 15
W: WAIT_EOC · 66

AIn-32/16 Rev. B

A: ADC · 18
C: CHECKLED · 4
R: READADC · 38
READADC_SCONV · 39
S: SETLED · 8
SET_MUX · 46
SE_DIFF · 44
START_CONV · 62
SYNCALL · 11
SYNCENABLE · 13
SYNCSTAT · 15
W: WAIT_EOC · 66

AIn-32/16 Rev. C

A: ADC · 18
C: CHECKLED · 4
R: READADC · 38
READADC_SCONV · 39
S: SEQ_MODE · 48
SEQ_READ · 50
SEQ_READ32 · 56
SEQ_READ_ONE · 51
SEQ_READ_PACKED · 54
SEQ_READ_TWO · 52
SEQ_SELECT · 57
SEQ_SET_DELAY · 58
SEQ_STATUS · 60
SETLED · 8
SET_MUX · 46
SE_DIFF · 44
START_CONV · 62
SYNCALL · 11
SYNCENABLE · 13
SYNCSTAT · 15
W: WAIT_EOC · 66

AIn-8/12 Rev. A

A: ADC · 18
C: CHECKLED · 4
R: READADC · 38
S: SETLED · 8
SET_MUX · 46
START_CONV · 62
SYNCALL · 11
SYNCENABLE · 13
SYNCSTAT · 15
W: WAIT_EOC · 66

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

AIn-F-4/12 Rev. A

AIn-F-4/14

AIn-F-4/14 Rev. A

AIn-F-4/16

AIn-F-4/16 Rev. A

AIn-F-8/12

AIn-F-8/12 Rev. A

AIn-F-8/14

AIn-F-8/14 Rev. A

AIn-F-8/16

AIn-F-8/16 Rev. A

AO-16/8-12

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

COMP16 Rev. A

CPU-T10

CPU-T9

DIO-32

DIO-32 Rev. B

Inter-SL

OPT-16 Rev. A

OPT-16 Rev. B

Profi-SL

PT100-4, PT100-8

PWM-4(-I)

REL-16 Rev. A

REL-16 Rev. B

RS232-2, RS232-4

RS485-2, RS485-4

Storage Rev. A

TC-16

TC-4

TC-8

TRA-16 Rev. A

TRA-16 Rev. B

(LP)SH-8(-FI)

A-3 Thematic Instruction List

The instructions are divided into the following groups. Inside a group the instructions are sorted alphabetically.

- Analog inputs
- Analog outputs
- Communication interface
  - CAN
  - Fieldbus
  - RSxxx
- Counters
- Data Storage
- Digital in-/outputs
- System
- Temperature inputs

Analog Inputs

ADC	The instruction ADC executes a complete measurement process on a 12-bit, 14-bit or 16-bit ADC.
ADC16	The instruction ADC16 executes a complete measurement on a 16-bit ADC. The information apply only for the module Pro-AIn-8/16 REVA.
ADCF	The instruction ADCF executes a complete measurement on a Fast-ADC.
BURST_ABORT	The instruction Burst_Abort aborts a running burst-measurement sequence on the specified module.
BURST_CREAD	The instruction 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	The instruction Burst_CStart starts a burst-measurement sequence in the "Continuous" mode on the specified module.
BURST_INIT	The instruction Burst_Init sets the parameters for a burst-measurement sequence on the specified module.
BURST_READ	The instruction Burst_Read copies the measurement values of a channel into a specified array.
BURST_READ_PACKED	The instruction 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	The instruction Burst_Start starts (independent of the processor) a burst-measurement sequence on the specified module.
BURST_STATUS	The instruction Burst_Status determines the amount of the burst-measurements which are still to be executed on the specified module.
READADC	The instruction ReadADC reads the result of a conversion from the ADC register of the specified module.
READADCF	The instruction ReadADCF reads out the conversion result from an F-ADC of the specified module.
READADCF_32	The instruction 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	The instruction ReadADCF_SConv reads out the conversion result from an F-ADC of the specified module and starts immediately a new conversion.
READADCF_SCONV_32	The instruction 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	The instruction ReadADC_SConv reads out the conversion result from an ADC of the specified module and starts immediately a new conversion.
SEQ_MODE	The instruction 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	The instruction Seq_Read copies a specified amount of measurement values (16-bit each) from the module to a destination array.
SEQ_READ32	The instruction Seq_Read32 copies all 32 measurement values (16-bit each) from the specified module into the destination array.
SEQ_READ_ONE	The instruction Seq_Read_One reads out a specified measurement value (16 bit) of a measurement group on the specified module.
SEQ_READ_PACKED	The instruction 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	The instruction Seq_Read_Two reads out at the same time 2 consecutive measurement values (16-bit each) of a measurement group, on the specified module and returns them in a 32-bit value.
SEQ_SELECT	The instruction Seq_Select determines the channels belonging to the measurement group, which will be converted by the sequential control on the specified module.
SEQ_SET_DELAY	The instruction Seq_Set_Delay determines the settling time (waiting time between 2 measurements) of the sequential control on the specified module.
SEQ_STATUS	The instruction Seq_Status determines how many channels of the measurement group are already converted and stored by the sequential control of the specified module.
SET_GAIN	The instruction Set_Gain sets the operating mode for a channel of the specified module, and thus the gain factor and measurement range, too.

SET_MUX	The instruction Set_Mux sets the multiplexer input of the module to a specified channel and gain.
SE_DIFF	The instruction SE_Diff sets the operating mode single ended or differential for all analog inputs on the specified module.
SH_SETMODE	The instruction SH_SetMode sets the mode of the sample and hold levels.
START_CONV	The instruction Start_Conv starts the A/D conversion on the specified module.
START_CONVF	The instruction Start_ConvF starts the conversion of one or more F-ADCs of the specified module.
SYNC_MODE	The instruction Sync_Mode determines on the specified module the type of synchronization with other modules, especially for burst-measurement sequences.
WAIT_EOC	The instruction Wait_EOC waits, until the last A/D conversion has finished.
WAIT_EOCF	The instruction Wait_EOCF waits until the end of conversion on all specified F-ADCs.

Analog Outputs

DAC	The instruction DAC outputs an (analog) voltage on the channel of the specified module, that corresponds to the indicated digital value.
FG_CONTROL	The instruction 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, the instruction FG_DEF defines the start address and the size of the internal buffer for the function generator mode.
FG_DELAY	The instruction FG_DELAY sets the output rate of function generator on the specified module.
FG_MODE	The instruction FG_MODE enables or disables the function generator mode on the specified module.
FG_READ	Die Anweisung FG_READ überträgt eine gerade Zahl von Daten aus dem internen Zwischenspeicher des angegebenen Moduls in ein Feld.
FG_READ_INDEX	The instruction FG_READ_INDEX returns the position pointer of a specified function generator.
FG_STATUS	The instruction FG_STATUS returns the status of all function generators of the module.
FG_WRITE	The instruction FG_Write transfers an even number of data of an array to a specified address in the buffer of the module.
START_DAC	The instruction Start_DAC starts the conversion or output of all DACs on the specified module.
WRITEDAC	The instruction WriteDAC writes a digital value into the output register of a DAC on the specified module. The conversion into output voltage is started by the instruction START_DAC.

Communication interface

CAN

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.
EN_RECEIVE	The instruction En_Receive enables a message object on the specified module to receive messages.
EN_TRANSMIT	The instruction En_Transmit enables a message object on the specified module to transmit messages.
GET_CAN_REG	The instruction Get_CAN_Reg returns the contents of a specified register on a CAN controller on the specified module.
INIT_CAN	The instruction Init_CAN initializes one of the CAN controllers on the specified module and sets it into an initial status.
READ_MSG	The instruction 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	The instruction Set_CAN_Baudrate sets the baud rate on one of the controllers on the specified module and returns the information if this was successful.
SET_CAN_REG	The instruction Set_CAN_Reg writes a value in a register of the selected CAN controller on the specified module.

TRANSMIT	The instruction Transmit reads the data from the array CAN_MSG. As soon as the message object in one of the CAN controllers has access rights to the CAN bus, the message is sent.

Fieldbus

CHANGED_DATA	The function 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	The function Check_Access returns to which area of the DP-RAM the application has access rights.
GET_PRO_BYTE	The function Get_Pro_Byte returns a byte of a specified memory address of the DP-RAM of the fieldbus module.
GET_READ_BUFFER	The instruction Get_Read_Buffer copies a defined data block from the memory area of the DP-RAM into the specified destination array.
INIT_SLAVE	The instruction Init_Slave initializes the fieldbus slave and can only be used after power up.
REQUEST_ACCESS	The instruction Request_Access requests access to the DP-RAM of the slave.
REQUEST_RELEASE_ACCESS	The instruction Request_Release_Access requests to return the access right for the DP-RAM of the slave to the fieldbus.
SET_PRO_BYTE	The instruction Set_Pro_Byte sets a byte in the DP-RAM of the fieldbus slave.
SET_WRITE_BUFFER	The instruction Set_Write_Buffer copies the data from an array into a specified memory area of the DP-RAM.

RSxxx

CHECK_SHIFT_REG	The instruction 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	The instruction GET_RS reads out the controller register on the specified module.
READ_FIFO	The instruction READ_FIFO reads a value from the input FIFO of a specified channel on the specified module.
RS485_SEND	The instruction RS485_Send determines the transfer direction for a specified channel on the specified module.
RS_INIT	The instruction RS_INIT initializes one channel on the specified module.
RS_RESET	The instruction RS_RESET executes a hardware reset on the specified module and deletes the settings for all channels.
SET_RS	The instruction SET_RS writes a value into a specified register on the specified module.
WRITE_FIFO	The instruction WRITE_FIFO writes a value into the send-FIFO of a specified channel on the specified module.

Comparator

COMP_DIGIN_WORD	The instruction COMP_DIGIN_WORD returns the current status of the threshold value comparison for all channels of the specified module.
COMP_DIGIN_WORD_DIFF	The instruction 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	The instruction 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	The instruction COMP_FIFO_SELECT determines the channel pair whose data is stored in the internal FIFO memory of the module.
COMP_READ	The instructions COMP_READ returns the current minimum or maximum measurement value of a specified channel on the module.
COMP_RESET	The instruction COMP_RESET resets the measurement of the minimum and maximum values simultaneously for the selected channels.
COMP_SET	The instruction COMP_SET determines the lower and upper threshold value for a specified channel.

Counters

CNT_CLEAR	The instruction Cnt_Clear sets the counter values of one or more counters on the specified module to the value 0 (zero).
CNT_ENABLE	The instruction Cnt_Enable enables or disables one or more counters on the specified module.
CNT_LATCH	The instruction 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	The instruction Cnt_Read16 returns the current counter value of a 16-bit counter on the specified module.
CNT_READ32	The instruction Cnt_Read32 returns the current counter value of a 32-bit counter on the specified module.
CNT_READLATCH16	The instruction Cnt_ReadLatch16 returns the value from the latch register of a 16-bit counter on the specified module.
CNT_READLATCH32	The instruction Cnt_ReadLatch32 returns the value from the latch register of a 32-bit counter on the specified module.
CNT_SETMODE	The instruction Cnt_SetMode sets the operating mode of all counters on the specified module, four edge evaluation or clock and direction input.
CO4_CLEARENABLE	The instruction CO4_ClearEnable enables the external input CLR of one or more counters.
CO4_GETSTATUS	The instruction CO4_GetStatus returns the status of the input signals of a counter on the specified module as bit pattern.
CO4_LATCHENABLE	The instruction CO4_LatchEnable enables the external input LATCH of one or more counters on the specified module.
CO4_READ	The instruction CO4_Read returns the current counter value from the specified module.
CO4_READLATCH	The instruction CO4_ReadLatch returns the value of the latch register of a counter on the specified module.
CO4_RESETSTATUS	The instruction CO4_RESETsTATUS clears the status register of one or more counters on the specified module.
CO4_SETMODE	The instruction CO4_SetMode sets the count mode of a counter on the specified module.
CO4_SET_LATCHMODE	The instruction CO4_sET_lATCHmODE determines the mode of the latch-inputs for all counters on the specified module.
EXTLCH_ENABLE	The instruction ExtLch_Enable enables or disables all latch-inputs on the specified module. The latch-inputs are selected by the corresponding counter number.
SSI_MODE	The instruction SSI_MODE sets the modes of all SSI decoders on the specified module, either "single shot" (read out once) or "continous" (read out continously).
SSI_READ	The instruction SSI_READ returns the last saved counter value of a specified SSI counter on the specified module.
SSI_SET_BITS	The instruction 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	The instruction 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	The instruction SSI_START starts the reading of one or both SSI encoders on the specified module (only in mode "single shot").
SSI_STATUS	The instruction SSI_Status returns the current read-status on the speicified module for a specified decoder.

Data Storage

MEDIA_RD_BLK_F	The instruction 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	The instruction 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	The instruction 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	The instruction 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	The instruction 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.
RTC_GET	The instruction RTC_GET returns date and time from the real-time clock of the specified module. Invalid values are not accepted.
RTC_SET	The instruction RTC_GET returns date and time from the real-time clock of the specified module. Invalid values are not accepted.

Digital Inputs/Outputs

DIGIN_LONG_F	The instruction Digin_Long_F returns the status of the inputs (bits 31…00) of the specified module as bit pattern.
DIGIN_WORD1	The instruction Digin_Word1 returns the status of the inputs 0…15 of the specified module as bit pattern.
DIGIN_WORD2	The instruction Digin_Word2 returns the status of the inputs 16…31 of the specified module as bit pattern.
DIGOUT	The instruction Digout sets a single output of the specified module to the level "high" or "low". All other outputs remain unchanged.
DIGOUT_BITS_F	The instruction Digout_Bits_F sets the specified outputs of the specified module to the levels "high" or "low".
DIGOUT_F	The instruction Digout_F sets a single output of the specified module to the level "high" or "low". All other outputs remain unchanged.
DIGOUT_LONG_F	With the 32-bit value the instruction Digout_Long_F sets or clears all outputs on the specified module.
DIGOUT_WORD1	The instruction Digout_Word1 sets the digital outputs 0…15 on the specified module simultaneously to the specified levels.
DIGOUT_WORD2	The instruction Digout_Word2 sets the digital outputs 16…31 on the specified module simultaneously to the specified levels.
DIGPROG1	The instruction DIGPROG1 programs the digital channels 0…15 of the specified module as input or output.
DIGPROG2	The instruction DIGPROG2 programs all channels 16…31 of the specified module as inputs or outputs.
DIG_LATCH	The instruction 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	The instruction Dig_ReadLatch1 returns the lower 16 bits (bit 0…bit 15) from the latch register for the digital inputs of the specified module.
DIG_READLATCH2	The instruction 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	The instruction 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	The instruction 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	The instruction Dig_WriteLatch32 writes a 32-bit value into the long-word (bits 31…0) of the latch on the specified module.
GET_DIGOUT_LONG	The instruction Get_Digout_Long returns the contents of the output-latch (register for digital outputs) on the specified module.
GET_DIGOUT_WORD1	The instruction 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	The instruction GET_DIGOUT_WORD2 returns the upper word (bits 16…31) of the output-latch (register for digital outputs) of the specified module.
PWM_ENABLE	The instruction PWM_ENABLE enables or disables all internal counters of the specified module. The counters are selected by the corresponding PWM output number.
PWM_OUT	The instruction PWM_Out sets a specified PWM output channel on the specified module to the level "high" or "low".
PWM_SET	The instruction PWM_SET makes the settings for a specified PWM output channel on the specified module.

System

CHECKLED	The instruction CheckLED returns the status of the green LED (on top of the front panel) of the module.
CPU_DIGIN	The instruction 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	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	The instruction RESETWATCHDOGTIMER sets the watchdog counter of the CPU module to the start value. The counter remains enabled.
SETLED	The instruction SETLED switches the green LED (on top of the front panel) on or off.
STARTWATCHDOG	The instruction StartWATCHDOG activates the watchdog counter of the CPU module and sets its start value.
STOPWATCHDOG	The instruction StopWATCHDOG disables the watchdog counter of the CPU module.
SYNCALL	The instruction SyncAll starts a specified action synchronically on all modules which have been activated before with SyncEnable.
SYNCENABLE	The instruction SyncEnable enables or disables the synchronizing option on the specified module.
SYNCSTAT	The instruction Syncstat returns the settings of the synchronizing option of the specified module.

Temperature-Inputs

TC_SELECT	The instruction TC_select sets the thermocouple channels via multiplexer to the analog output of the module.