LIBAD4

Library for Programming Interface LIBAD4

Programming Guide

Version 5.0

Contents

1 Overview 1
1.1 Introduction 1
1.2 Copyrights 2
2 Installation 3
2.1 Installation under Windows® 3
2.2 Sharing the Library 3
3 Basics 4
3.1 General 4
4 Single-Value Acquisition 6
4.1 Function Description (Single Values) 6
4.1.1 ad_open 6
4.1.2 ad_close 8
4.1.3 ad_get_range_count 8
4.1.4 ad_get_range_info 9
4.1.5 ad_discrete_in 9
4.1.6 ad_discrete_in64 11
4.1.7 ad_discrete_inv 11
4.1.8 ad_discrete_out 12
4.1.9 ad_discrete_out64 13
4.1.10 ad_discrete_outv 14
4.1.11 ad_sample_to_float 16
4.1.12 ad_sample_to_float64 16
4.1.13 ad_float_to_sample 17
4.1.14 ad_float_to_sample64 18
4.1.15 ad_analog_in 18
4.1.16 ad_analog_out 19
4.1.17 ad_digital_in 19
4.1.18 ad_digital_out 19
4.1.19 ad_set_digital_line 19
4.1.20 ad_get_digital_line 20
4.1.21 ad_get_line_direction 20
4.1.22 ad_set_line_direction 20
4.1.23 ad_get_version 20
4.1.24 ad_get_drv_version 21
4.1.25 ad_get_product_info 21
5 Scan Process 22
5.1 Notes 22
5.2 Scan Parameters 22

© BMC Messsysteme GmbH

5.2.1 struct ad_scan_cha_desc 22
5.2.1.1 Speicherarten 23
5.2.1.2 Trigger Types 24
5.2.2 struct ad_scan_desc 24
5.2.3 struct ad_scan_state 26
5.2.4 struct ad_scan_pos 26
5.2.5 struct ad_cha_layout 27
5.3 Memory-only Scan 27
5.3.1 Starten eines Scans 27
5.3.2 Reading out Measuring Values 28
5.3.3 Stopping a Scan 29
5.4 Continuous Scan 30
5.4.1 Composition of a RUN 30
5.4.2 One Sample per RUN 33
5.4.3 Signals with Different Storage Ratio 34
5.5 Scan with Triggering 35
5.5.1 Parameters for scan channels with triggering 35
5.6 Functions Description (Scan) 38
5.6.1 ad_start_mem_scan 38
5.6.2 ad_start_scan 40
5.6.3 ad_get_sample_layout 40
5.6.4 ad_get_samples 41
5.6.5 ad_get_samples_f 41
5.6.6 ad_get_samples_f64 42
5.6.7 ad_calc_run_size 43
5.6.8 ad_get_next_run 43
5.6.9 ad_get_next_run_f 44
5.6.10 ad_get_next_run_f64 44
5.6.11 ad_poll_scan_state 45
5.6.12 ad_stop_scan 45

6 Data Acquisition Systems 46

6.1 Notes 46
6.2 LAN-AD16fx / AMS42/84-LAN16fx 46
6.2.1 Channel Numbers LAN-AD16fx / AMS42/84-LAN16fx 46
6.2.2 Configuration of the LAN-AD16fx / AMS42/84-LAN16fx Counters 48
6.3 PCIe-BASE / PCI-BASEII / PCI-PIO 51
6.3.1 Digital Ports and Counters 51
6.3.1.1 PCIe-BASE / PCI-BASEII / PCI-PIO 51
6.3.1.2 Configuration of the Counters 52
6.3.2 Plug-on Modules 54
6.3.2.1 MADDA16/16n 55

6.3.2.2 MDA16-4i/-8i....55
6.3.2.3 Functionsgenerator of the MDA16-4i/-8i....56
6.4 USB-AD / USB-AD-OEM / USB-PIO / USB-PIO-OEM....60
6.4.1 Key Data / Channel Numbers USB-AD....60
6.4.2 Key Data / Channel Numbers USB-PIO(-OEM)....61
6.5 USB-AD14f....61
6.5.1 Key Data / Channel Numbers USB-AD14f....61
6.6 USB-AD16f / AMS42/84-USB....62
6.6.1 Key Data / Channel Numbers USB-AD16f / AMS42/84-USB....63
6.7 USB-OI16....63
6.7.1 Key Data / Channel Numbers USB-OI16....64
6.7.2 Channel numbers USB-OI16....64
6.7.3 Configuration of the USB-OI16 Counters....64

7 Index....67

© BMC Messsysteme GmbH

1 Overview

1.1 Introduction

The library **LIBAD4** is a programming interface to all data acquisition systems from BMC Messsysteme GmbH. This interface features reading and writing of single values, reading of an analog input or the output of a channel value, for example.

Besides the input and output of single values, it is possible to run a scan with the **LIBAD4**. Scanning the input channels is done in the relating driver and is time-decoupled from the application allowing for fast sampling of the input channels without loosing any measuring values.

The **LIBAD4** is provided for Windows® XP/7/8 as well as for Mac OS X, FreeBSD and Linux. That means that cross-platform use of the DAQ systems from BMC Messsysteme GmbH is possible without having to change the source code.

- **LibadX is a 32-bit interface. If programming on a 64-bit system, the application must be created as a 32-bit application.**
- **Please note, these code extracts as well as all the other examples in this manual consciously skip any error handling to simplify matters. Of course, this has to be realized in self-written programs.**

1.2 BMC Messsysteme GmbH

BMC Messsysteme GmbH stands for innovative measuring technology made in Germany. We provide all components required for the measuring chain, from sensor to software.

Our hardware and software components are perfectly tuned with each other to produce an extremely user-friendly integrated system. We put great emphasis on observing current industrial standards, which facilitate the interaction of many components.

Products by BMC Messsysteme are applied in industrial large-scale enterprises, in research and development and in private applications. We produce in compliance with ISO-9000-standards because standards and reliability are of paramount importance to us - for your profit and success.

Please visit us on the web (https://www.bmcm.de) for detailed information and latest news.

 © BMC Messsysteme GmbH

1.3 Copyrights

The programming interface **LIBAD4** with all extensions has been developed and tested with utmost care. BMC Messsysteme GmbH does not provide any guarantee in respect of this manual, the hard- and software described in it, its quality, its performance or fitness for a particular purpose. BMC Messsysteme GmbH is not liable in any case for direct or indirect damages or consequential damages, which may arise from improper operation or any faults whatsoever of the system. The system is subject to changes and alterations which serve the purpose of technical improvement.

The programming interface **LIBAD4**, the manual provided with it and all names, brands, pictures, other expressions and symbols are protected by law as well as by national and international contracts. The rights established therefrom, in particular those for translation, reprint, extraction of depictions, broadcasting, photomechanical or similar way of reproduction - no matter if used in part or in whole - are reserved. Reproduction of the programs and the manual as well as passing them on to others is not permitted. Illegal use or other legal impairment will be prosecuted by criminal and civil law and may lead to severe sanctions.

Copyright © 2014
Updated: 12/09/2014

BMC Messsysteme GmbH
Hauptstrasse 21
82216 Maisach
GERMANY
Phone: +49 8141/404180-1
Fax: +49 8141/404180-9
E-mail: info@bmcm.de

2 Installation

2.1 Installation under Windows®

Under Windows®, the **LIBAD4** is implemented as "dynamic link library". The installation program copies the library together with all the header files and the example programs to hard disc.

The `libad4.dll` should be copied into the relating program directory in order for the programs to access the library.

All functions of the LIBAD4 use the calling conventions *cdecl* of C. If working with the library under another programming language than C/C++, make sure it uses the calling conventions of C for the LIBAD4 functions.

2.2 Sharing the Library

The **LIBAD4** library must be installed on the target system for the provided functions to be available to an application. Therefore, sharing the following files is expressively permitted (if the version number of your **LIBAD4** differs, it must be adapted accordingly).

```
libad4.dll
libad4.dylib
libad4.so.4.6.523
```

It is the task of the application's installation program, to install the relevant file together with the application. The LIBAD4 SDK should certainly not be used to install the LIBAD4 library on the target machine.

Please note that all other files of the LIBAD4 SDK must not be shared!

© BMC Messsysteme GmbH

3 Basics

3.1 General

The functions exported by the **LIBAD4** and the used constants are available to a C/C++ program by the header file **`libad.h`**.

The precise definitions of the C/C++ commands and structures described in this manual are defined in the relevant header files.

The **LIBAD4** provides two functions to open or close the connection to a data acquisition system. A DAQ system is opened with the **`ad_open()`** function, the connection is closed with **`ad_close()`**. The following example demonstrates the basic procedure:

Prototyp

```
int32_t
ad_open (const char *name);
```

C

```
#include "libad.h"

...
int32_t adh;
...

  adh = ad_open ("usb-ad");
  if (adh == -1)
    {
      printf ("failed to open USB-AD driver\n");
      exit (1);
    }
...
  ad_close (adh);
```

The name of the data acquisition system is passed to the function **`ad_open()`**. This string is not case-sensitive, i.e. "usb-ad" and "USB-AD" both open the USB-AD. The function returns a handle required for all further calls of the **LIBAD4**. In case of an error, **`-1`** will be returned. On Windows®, the error number can be retrieved with **`GetLastError()`**.

Of course, it is also possible to open several data acquisition systems at the same time. In this case, **`ad_open()`** returns another handle for each open driver. Please see the description of the **`ad_open ()`** function (p. 6) for detailed information.

The supported DAQ systems, the channel numbers of the inputs and outputs and the permitted ranges are specified in chapter "Data Acquisition Systems" on page 46 and the following.

As soon as a data acquisition system has been opened, incoming measuring values at the inputs can be read in (see "**`ad_discrete_in`**", p. 9) or output values can be set (see "**`ad_discrete_out`**", p. 12). Please see the chapter about "Single-Value Acquisition" on page 6 for further details.

In addition to reading single measuring values, the **LIBAD4** can also start a scan. In this case, several input channels are periodically sampled and the recorded measuring values are written to a buffer. Programming a scan is described in chapter "Scan Process" on page 22 and the following.

If single measuring values are read out, one command per query is sent to the DAQ system. When programming a scan, one command is sent to the device at scan start only. Afterward, the DAQ system continuously sends measuring data.

As sending a command always implies a certain latency, single values can never be read out within the same time that is reached when programming a scan.

© BMC Messsysteme GmbH

4 Single-Value Acquisition

4.1 Function Description (Single Values)

The LIBAD4 functions are thread-safe unless otherwise expressly specified in the function description.

4.1.1 ad_open

Prototyp

```
int32_t
ad_open (const char *name);
```

C

```
#include "libad.h"

...
int32_t adh;
...

  adh = ad_open ("usb-ad");

  if (adh == -1)
    {
      printf ("failed to open USB-AD\n");
      exit (1);
    }

...

  ad_close (adh);
```

The **`ad_open()`** function provides a connection to the data acquisition system by passing the name of the device. The passed string is not case-sensitive, i.e. "**`pcibase`**" and "**`PCIbase`**" both open the PCIe-BASE / PCI-BASEII / PCI-PIO. The function returns a handle required for all further calls of the **LIBAD4**. In case of an error, **`-1`** will be returned. Under Windows®, the error number can be retrieved with **`GetLastError()`**.

Of course, it is also possible to open several data acquisition systems at the same time. In this case, **`ad_open()`** returns another handle for each open driver.

Hardware specific information (e.g. name of DAQ system) about the supported DAQ systems are provided in the respective chapters of the same name:

LAN-AD16fx / AMS42/84-LAN

PCIe-BASE / PCI-BASEII / PCI-PIO

MADDA16/16n / MDA16-4i/-8i

USB-AD14f / USB-AD16f / AMS42/84-USB / USB-OI16

USB-AD / USB-AD-OEM / USB-PIO / USB-PIO-OEM

The following example opens a USB-AD and a USB-PIO:

C

```
#include "libad.h"

...
int32_t adh1;
int32_t adh2;
...

  adh1 = ad_open ("usb-ad");
  adh2 = ad_open ("usb-pio");

...

  ad_close (adh1);
  ad_close (adh2);
```

To open several devices of the same type, the number of the data acquisition system, separated by a colon, is added to the name as a suffix. The following example opens two USB-AD units:

C

```
#include "libad.h"

...
int32_t adh1;
int32_t adh2;
...

  adh1 = ad_open ("usb-ad:0");
  adh2 = ad_open ("usb-ad:1");
...

  ad_close (adh1);
  ad_close (adh2);
```

Alternatively, a DAQ system can be opened with its serial number by entering the serial number with an @ character after the colon.

The following example opens the two USB-AD units with the serial numbers 157 and 158.

© BMC Messsysteme GmbH

C

```
#include "libad.h"

...
int32_t adh1;
int32_t adh2;
...

  adh1 = ad_open ("usb-ad:@157");
  adh2 = ad_open ("usb-ad:@158");
...

  ad_close (adh1);
  ad_close (adh2);
```

4.1.2 ad_close

Prototype

```
int32_t
ad_close (int32_t adh);

C#include "libad.h"

...
int32_t adh;
...

  adh = ad_open ("usb-ad");

  if (adh == -1)
    {
      printf ("failed to open USB-AD\n");
      exit (1);
    }
...

  ad_close (adh);
```

The **`ad_close()`** function shuts the connection to the data acquisition system. The function returns **`0`** or the relevant error number in case of an error.

4.1.3 ad_get_range_count

Prototype

```
int32_t
ad_get_range_count (int32_t adh, int32_t cha, int32_t cnt);
```

The **`ad_get_range_count()`** function returns the number of measuring ranges of the channel **`cha`**.

4.1.4 ad_get_range_info

Prototype

```
struct ad_range_info
{
  double min;
  double max;
  double res;
...
  int bps;
  char unit[24];
};

int32_t
ad_get_range_info (int32_t adh, int32_t cha, int32_t range,
struct ad_range_info *info);
```

C

```
#include "libad.h"

...
int32_t adh;
int32_t cnt;
int32_t cha;
struct ad_range_info info;
...

  adh = ad_open ("usbbase");
  cha = AD_CHA_TYPE_ANALOG_IN;
  rc = ad_get_range_count(adh, cha, &cnt);
  for (i=0;i < cnt; i++)
    {
      rc = ad_get_range_info(adh, cha, i, &info);
...
    }
...
  ad_close (adh);
```

The **`ad_get_range_info()`** function returns the information of the measuring range **`range`** of the channel **`cha`**.

4.1.5 ad_discrete_in

Prototype

```
int32_t
ad_discrete_in (int32_t adh, int32_t cha,
                int32_t range, uint32_t *data);
```


© BMC Messsysteme GmbH

C

```
int32_t adh;
int32_t st;
uint32_t data;

...

  adh = ad_open ("usb-ad");

  st = ad_discrete_in (adh, AD_CHA_TYPE_ANALOG_IN|1,
                        0, &data)
...
  ad_close (adh);
```

The **`ad_discrete_in()`** function returns a single value of the specified channel. In addition to the channel number, the channel type is also entered as parameter:

AD_CHA_TYPE_ANALOG_IN	for analog inputs
AD_CHA_TYPE_ANALOG_OUT	for analog outputs
AD_CHA_TYPE_DIGITAL_IO	for digital channels
AD_CHA_TYPE_COUNTER	for counter channels

Depending on the DAQ system, different channels are available. These are specified in chapter "Data Acquisition Systems" (p. 46). Besides the channel number, the measuring range used for sampling the input channel is passed to the function. This does not apply to digital channels.

For analog channels, the **`ad_discrete_in()`** function returns a value between **`0x00000000`** and **`0xffffffff`** in **`*data`**. The value **`0x00000000`** relates to the lower range limit, the value **`0x100000000`** is the upper range limit (this value is not reached at 32-bit returning **`0xffffffff`** at the maximum). The value **`0x80000000`** is equivalent to the middle of the range, i.e. 0.0V for a symmetric, bipolar input.

To convert such a value into a voltage value, the **`ad_discrete_out64()`** function is provided. The auxiliary function **`ad_analog_in()`** directly passes the sampled value as voltage.

Channel number and range number depend on the DAQ hardware used and are documented in the relating chapters ((see "Data Acquisition Systems", p. 46).

4.1.6 ad_discrete_in64

Prototype

```
int32_t
ad_discrete_in64 (int32_t adh, int32_t cha,
                  uint64_t range, uint64_t *data)

Cint32_t adh;
int32_t st;
uint64_t data;

...

  adh = ad_open ("usb-ad");

  st = ad_discrete_in64 (adh, AD_CHA_TYPE_ANALOG_IN|1,
                         0, &data)

...

  ad_close (adh);
```

The **`ad_discrete_in64()`** function returns a single value of the specified channel. Besides the channel number, the measuring range used for sampling the input channel is passed to the function. This does not apply to digital channels.

The **`ad_discrete_in64()`** function returns a value between **`0x0000000000000000`** (lower range limit) and **`0x10000000000000000`** (upper range limit). The entire 64-bit range is only used by special 64-bit DAQ systems. The value **`0x8000000000000000`** is equivalent to the middle of the range, i.e. 0.0V for a symmetric, bipolar input.

To convert such a value into a voltage value, the **`ad_sample_to_float64()`** function is provided. The auxiliary function **`ad_analog_in()`** directly passes the sampled value as voltage.

Channel number and range number depend on the DAQ hardware used and are documented in the relating chapters (see "Data Acquisition Systems", p. 46).

4.1.7 ad_discrete_inv

Prototype

```
int32_t
ad_discrete_inv (int32_t adh, int32_t chac,
                 int32_t chav[], uint64_t rangev[],
                 uint64_t datav[]);
```


© BMC Messsysteme GmbH

C

```
#define CHAC 3

uint64_t rangev[CHAC], datav[CHAC];
int32_t chav[CHAC], adh, i;

/* das Beispiel liest die 3 Kanäle der USB-PIO
 */

adh = ad_open ("usb-pio");
if (adh < 0)
  {
    fprintf (stderr, "error: couldn't open USB-PIO\n");
    return -1;
  }

/* setze den range bei allen Kanälen auf 0 */
memset (rangev, 0, sizeof(*rangev));
for (i = 0; i < CHAC; i++)
  {
    /* Kanalnummer setzen */
    chav[i] = AD_CHA_TYPE_DIGITAL_IO|(i+1);
    /* auf Eingang setzen */
    ad_set_line_direction (adh, chav[i], 0xffffffff);
  }

ad_discrete_inv (adh, CHAC, chav, rangev, datav);
ad_close (adh);
```

The **`ad_discrete_inv()`** function reads **`chac`** inputs at once no matter if analog or digital. In addition to the channel numbers, the input ranges are passed to the function.

The routine **`ad_discrete_inv()`** is processed a little bit faster normally than the repeated call of the **`ad_discrete_in64()`** function in an appropriate loop.

Unlike **`ad_discrete_in()`** and **`ad_discrete_in64()`**, channel numbers, measuring ranges and value variables are passed to **`ad_discrete_inv()`** by arrays. The array values are set analogous to the **`ad_discrete_in64()`** function.

4.1.8 ad_discrete_out

Prototype

```
int32_t
ad_discrete_out (int32_t adh, int32_t cha,
                  int32_t range, uint32_t data);
```

C

```
int32_t adh;
int32_t st;
...

  adh = ad_open ("usb-ad");

  st = ad_discrete_out (adh, AD_CHA_TYPE_ANALOG_OUT|1,
                        0, 0x80000000)
...

  ad_close (adh);
```

The **`ad_discrete_out()`** function sets an output. Besides the channel number, the output range is passed to the function (only applies to DAQ systems with output ranges programmable via software). Otherwise, it has to be ensured by means of software that the specified output range conforms to the hardware settings.

As is the case with an analog input, the value **`0x00000000`** of an analog output relates to the lowest output voltage. The value **`0x100000000`** is the highest output voltage (this value is not reached at 32-bit so that **`0xffffffff`** at the maximum can be passed to **`ad_discrete_out()`**).

To convert a voltage value (float) to a digital value, which is passed to **`ad_discrete_out()`**, the **`ad_float_to_sample()`** function is provided. The auxiliary function **`ad_analog_out()`** directly passes the measured value as voltage.

Channel number and range number depend on the DAQ hardware used and are documented in the relating chapters (see "Data Acquisition Systems", p. 46).

4.1.9 ad_discrete_out64

Prototype

```
int32_t ad_discrete_out64 (int32_t adh, int32_t cha,
                   uint64_t range, uint64_t data);
```

 © BMC Messsysteme GmbH

C

```
int32_t adh;
int32_t st;
uint64_t data;

...

  adh = ad_open ("pcibase");

  st = ad_float_to_sample64 (adh,
                             AD_CHA_TYPE_ANALOG_OUT|1,
                             0, 0.0f, &data);
...

  st = ad_discrete_out (adh,
                        AD_CHA_TYPE_ANALOG_OUT|1,
                        0, data)
...

  ad_close (adh);
```

The **ad_discrete_out()** function sets an output. Besides the channel number, the output range is passed to the function (only applies to DAQ systems with output ranges programmable via software). Otherwise, it has to be ensured by means of software that the specified output range conforms to the hardware settings.

As is the case with an analog input, the value **0x0000000000000000** of an analog output relates to the lowest output voltage. The value **0x10000000000000000** is the highest output voltage. The entire 64-bit range of **ad_discrete_out64()** is only used by special 64-bit DAQ systems.

To convert a voltage value (float) to a digital value, which is passed to **ad_discrete_out64()**, the **ad_float_to_sample64()** function is provided. The auxiliary function **ad_analog_out()** directly passes the measured value as voltage.

Channel number and range number depend on the DAQ hardware used and are documented in the relating chapters (see "Data Acquisition Systems", p. 46).

4.1.10 ad_discrete_outv

Prototype

```
int32_t
ad_discrete_outv (int32_t adh, int32_t chac,
                  int32_t chav[], uint64_t rangev[],
                  uint64_t datav[]);
```

C

```
#define CHAC 3

uint64_t rangev[CHAC], datav[CHAC];
int32_t chav[CHAC], adh, i;

/* das Beispiel setzt die 3 Digitalports der USB-PIO
 * auf die Werte 1, 2 und 4 */

adh = ad_open ("usb-pio");
if (adh < 0)
  {
    fprintf (stderr, "error: couldn't open USB-PIO\n");
    return -1;
  }

/* setze den range bei allen Kanälen auf 0 */
memset (rangev, 0, sizeof(*rangev));
for (i = 0; i < CHAC; i++)
  {
    /* Kanalnummer setzen */
    chav[i] = AD_CHA_TYPE_DIGITAL_IO|(i+1);
    /* auf Ausgang setzen */
    ad_set_line_direction (adh, chav[i], 0);
    /* Wert setzen */
    datav[i] = 1 << i;
  }

ad_discrete_outv (adh, CHAC, chav, rangev, datav);
ad_close (adh);
```

The **`ad_discrete_outv()`** function sets **`chac`** outputs at once no matter if analog or digital. In addition to the channel numbers, the output ranges are passed to the function.

The routine **`ad_discrete_outv()`** is processed a little bit faster normally than the repeated call of the **`ad_discrete_out64()`** function in an appropriate loop.

Unlike **`ad_discrete_out()`** and **`ad_discrete_out64()`**, channel numbers, output ranges and values are passed to **`ad_discrete_outv()`** by arrays. The array values are set analogous to the **`ad_discrete_out64()`** function.

 © BMC Messsysteme GmbH

4.1.11 ad_sample_to_float

```
Prototype

int32_t
ad_sample_to_float (int32_t adh, int32_t cha,
                    int32_t range, uint32_t data
                    float *f);

Cint32_t adh;
int32_t st, cha, range;
uint32_t data;
float volt;
...
  adh = ad_open ("usb-ad");
...
  cha = AD_CHA_TYPE_ANALOG_IN|1;
  range = 0;

  st = ad_discrete_in (adh, cha, range, &data)
  if (st == 0)
    st = ad_sample_to_float (adh, cha, range, data,
                             &volt)
...
  ad_close (adh);
```

Converts a measuring value into the respective voltage value.

Channel number and range number depend on the DAQ hardware used and are documented in the relating chapters (see "Data Acquisition Systems", p. 46).

4.1.12 ad_sample_to_float64

```
Prototype

int32_t
ad_sample_to_float64 (int32_t adh, int32_t cha,
                      uint64_t range, uint64_t data
                      double *dbl);
```

C

```
int32_t adh;
int32_t st, cha, range;
uint64_t data;
float volt;
...
  adh = ad_open ("usb-ad");
...
  cha = AD_CHA_TYPE_ANALOG_IN|1;
  range = 0;

  st = ad_discrete_in64 (adh, cha, range, &data);
  if (st == 0)
    st = ad_sample_to_float (adh, cha, range, data,
                             &volt);
...
  ad_close (adh);
```

Converts a measuring value into the respective voltage value.

Channel number and range number depend on the DAQ hardware used and are documented in the relating chapters (see "Data Acquisition Systems", p. 46).

4.1.13 ad_float_to_sample

Prototype

```
int32_t
ad_float_to_sample (int32_t adh, int32_t cha,
                    int32_t range, float f,
                    uint32_t *data);
```

C

```
int32_t adh;
int32_t st, cha, range;
uint32_t data;
...

  adh = ad_open ("usb-ad");

...

  cha = AD_CHA_TYPE_ANALOG_OUT|1;
  range = 0;

  st = ad_float_to_sample (adh, cha, range, 3.2,
                           &data);
  if (st == 0)
    st = ad_discrete_out (adh, cha, range, data);
...

  ad_close (adh);
```

 © BMC Messsysteme GmbH

Converts a voltage value into the respective measuring value.

Channel number and range number depend on the DAQ hardware used and are documented in the relating chapters (see "Data Acquisition Systems", p. 46).

4.1.14 ad_float_to_sample64

Prototype

```
int32_t
ad_float_to_sample64 (int32_t adh, int32_t cha,
                      uint64_t range, double dbl,
                      uint64_t *data);
```

C

```
int32_t adh;
int32_t st, cha, range;
uint64_t data;

...

  adh = ad_open ("usb-ad");

...

  cha = AD_CHA_TYPE_ANALOG_OUT|1;
  range = 0;

  st = ad_float_to_sample64 (adh, cha, range, 3.2,
                             &data)
  if (st == 0)
    st = ad_discrete_out64 (adh, cha, range, data)

...

  ad_close (adh);
```

Converts a voltage value into the respective measuring value.

Channel number and range number depend on the DAQ hardware used and are documented in the relating chapters (see "Data Acquisition Systems", p. 46).

4.1.15 ad_analog_in

Prototype

```
int32_t
ad_analog_in (int32_t adh, int32_t cha,
              int32_t range, float *volt);
```

This auxiliary function calls **`ad_discrete_in()`** and then converts the measured value into the voltage value using **`ad_discrete_out64()`**. Only analog inputs are supported, i.e. **`AD_CHA_TYPE_ANALOG_IN|cha`** is internally used as channel number.

Channel number and range number depend on the DAQ hardware used and are documented in the relating chapters (see "Data Acquisition Systems", p. 46).

4.1.16 ad_analog_out

Prototype

```
int32_t
ad_analog_out (int32_t adh, int32_t cha,
               int32_t range, float volt);
```

This auxiliary function converts the voltage value with **ad_float_to_sample()** and then calls **ad_discrete_out()**. Only analog outputs are supported, i.e. **AD_CHA_TYPE_ANALOG_OUTIN|cha** is internally used as channel number.

Channel number and range number depend on the DAQ hardware used and are documented in the relating chapters (see "Data Acquisition Systems", p. 46).

4.1.17 ad_digital_in

Prototype

```
int32_t
ad_digital_in (int32_t adh,
               int32_t cha, uint32_t *data);
```

This auxiliary function calls **ad_discrete_in()** with the channel number **AD_CHA_TYPE_DIGITAL_IO|cha**.

4.1.18 ad_digital_out

Prototype

```
int32_t
ad_digital_out (int32_t adh,
                int32_t cha, uint32_t data);
```

This auxiliary function calls **ad_discrete_out()** with the channel number **AD_CHA_TYPE_DIGITAL_IO|cha**.

4.1.19 ad_set_digital_line

Prototype

```
int32_t
ad_set_digital_line (int32_t adh, int32_t cha,
                     int32_t line, uint32_t flag);
```

This auxiliary function reads channel **AD_CHA_TYPE_DIGITAL_IO|cha** and then sets line number **line** according to the parameter **flag**. If **flag** is 0, the line will be reset. If **flag** is not equal to 0, the line will be set. The first line of a digital channel starts with 0.

© BMC Messsysteme GmbH

4.1.20 ad_get_digital_line

Prototype

```
int32_t
ad_get_digital_line (int32_t adh, int32_t cha,
                     int32_t line, uint32_t *flag);
```

This auxiliary function reads channel **AD_CHA_TYPE_DIGITAL_IO|cha** and then sets **flag** according to the line **line**. If the line is low, **flag** will be set to 0, otherwise to 1. The first line of a digital channel starts with 0.

4.1.21 ad_get_line_direction

Prototype

```
int32_t
ad_get_line_direction (int32_t adh, int32_t cha,
                       uint32_t *mask);
```

Returns a bit mask describing the direction of the digital line. Each set bit stands for an input line, each deleted bit for an output line. Bit #0 specifies the direction of the first line of the digital port.

4.1.22 ad_set_line_direction

Prototype

```
int32_t
ad_set_line_direction (int32_t adh, int32_t cha,
                       int32_t mask);
```

Sets the input or output direction for all lines of a digital channel **cha** by passing a bitmask describing the direction of the digital line. Each set bit defines an input line, each deleted bit an output line. Bit #0 specifies the direction of the first line of the digital port.

0xFFFF, for example, sets all digital lines to input, **0x0000** to output.

Please note some DAQ systems do not feature changing the direction of single lines or only provide hard-wired digital channels (e.g. digital port of USB-AD14f).

4.1.23 ad_get_version

Prototype

```
uint32_t
ad_get_version ();
```

Returns the version of the LIBAD4.DLL. This ID can be split with the macros **AD_MAJOR_VERS()**, **AD_MINOR_VERS()** and **AD_BUILD_VERS()**.

© BMC Messsysteme GmbH

4.1.24 ad_get_drv_version

Prototype

```
int32_t
ad_get_drv_version (int32_t adh, uint32_t *vers);
```

Returns the version of the DAQ card driver the **LIBAD4** is compatible with.

4.1.25 ad_get_product_info

Prototype

```
struct ad_product_info
{
..uint32_t serial;          /* serial number */
..uint32_t fw_version;      /* firmware version */
..char model[32];           /* model name */
..uint8_t res[256];         /* reserved */
};

int32_t
ad_get_product_info (int32_t adh, int id,
                     struct ad_product_info *info,
                     int32_t size);
```

The function **`ad_get_product_info()`** returns the serial number, firmware version, and the product name of the DAQ system opened with **`ad_open`**.

If the parameter id = 0 is used, the information of the opened DAQ system will be returned. Using id = 1 or 2 product information of a DAQ module integrated in the DAQ system can be retrieved (e.g. MADDA16 with PCIe-BASE).

© BMC Messsysteme GmbH

5 Scan Process

5.1 Notes

Besides single-value acquisition of measuring values, the **LIBAD4** can also start a scan process sampling several input channels in a constant time period and returning the recorded measuring values in a buffer.

The **LIBAD4** differs between so-called "memory-only" scans and continuous scans. A "memory-only" scan is so short that the whole measurement data of the scans can be stored in the main memory of the PC. The scan process is configured, started and the recorded data are provided in a buffer at the end of the scan.

A continuous scan returns the recorded measuring values to the caller block by block during the scan process. An internal memory management of the measuring data can be activated for DAQ systems which independantly run a scan (e.g. LAN-AD16fx / AMS42/84-LAN16fx, PCIe-BASE / PCI-BASEII / PCI-PIO with MAD/MADDA modules, USB-AD16f / AMS42/84-USB, USB-AD14f). Alternatively, an individual memory management can be realized. In both cases, the caller is responsible to read out and store the measuring data from the **LIBAD4** in time – otherwise it comes to an overrun of the samples and the scan process will be aborted.

5.2 Scan Parameters

The scan process is defined by means of the two structures **`struct ad_scan_desc()`** and **`struct ad_scan_cha_desc`**. Global parameters, such as sampling period and number of measuring values, are set in **`struct ad_scan_desc`**. The structure **`struct ad_scan_cha_desc`** specifying channel-specific data, like channel number or trigger settings, has to be filled out for each channel to be sampled.

5.2.1 struct ad_scan_cha_desc

The following source code shows the layout of **`struct ad_scan_cha_desc`**:

C

```c
struct ad_scan_cha_desc
{
  int32_t cha;
  int32_t range;
  int32_t store;
  int32_t ratio;
  uint32_t zero;
  int8_t trg_mode;
  ...
  uint32_t trg_par[2];
  int32_t samples_per_run;
  ...
};
```

The elements of the structure bear the following meaning:

`cha`

Determines the channel number to be sampled and recorded. The channel number depends on the hardware and is described in chapter "Data Acquisition Systems" (see p. 46).

`range`

Sets the measuring range of the channel. The number of the measuring range depends on the hardware and is described in chapter "Data Acquisition Systems" (see p. 46).

store

Defines together with **ratio** (see below) how the channel is to be stored. A detailed description of the storage types follows in the next chapter (see "Storage Types", p. 23).

ratio

Defines the storage interval (see "Storage Types", p. 23).

zero

Determines the zero level for RMS calculation. Only required if the root mean square value of the signal is to be stored.

trg_mode

Defines together with **trg_par[]** (see below) if and how this channel sets off a trigger.

trg_par[]

Defines the trigger levels.

samples_per_run

Is returned by the **LIBAD4** containing the number of measuring values produced for this channel.

Elements of the structure which are not used or documented must necessarily be set to 0!

5.2.1.1 Storage Types

Channels can be recorded in different ways. The storage type is defined by the **ratio** and **store** elements of the **struct ad_scan_cha_desc** structure.

The easiest case is to set **store** to **AD_STORE_DISCRETE** and **ratio** to **1**. Each recorded measuring value will be stored:

Time	0ms	2ms	4ms	6ms	8ms	10ms	12ms	14ms	16ms	18ms	20ms	22ms	...
Sample	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	...
Stored value	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	...

In addition to the recorded measuring value, also the mean value, minimum, maximum or RMS can be stored across an interval. This feature is provided by the **LIBAD4** by defining the following constants:

C

```c
#define AD_STORE_DISCRETE
#define AD_STORE_AVERAGE
#define AD_STORE_MIN
#define AD_STORE_MAX
#define AD_STORE_RMS
```

The table below illustrates the connection between the sampling rate and **ratio**. In this example, the sampling rate is 2msec and the mean value of channel **a** is stored with 1:5 ratio (i.e. **store** is set to **AD_STORE_AVERAGE** and **ratio** to 5).

Time	0ms	2ms	4ms	6ms	8ms	10ms	12ms	14ms	16ms	18ms	20ms	22ms	...

© BMC Messsysteme GmbH

Sample	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	...
Stored value					$\frac{1}{5}\sum a_i$					$\frac{1}{5}\sum a_i$			...

It is also possible to save several values created by different storage methods. The next example shows the storage of the last recorded value and the mean value of 5 measuring values (i.e. **`ratio`** is set to **`5`** and **`store`** to **`AD_STORE_DISCRETE|AD_STORE_AVERAGE`**):

Time	0ms	2ms	4ms	6ms	8ms	10ms	12ms	14ms	16ms	18ms	20ms	22ms	...
Sample	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	...
Stored value					a_5 $\frac{1}{5}\sum a_i$					a_{10} $\frac{1}{5}\sum a_i$			...

5.2.1.2 Trigger Types

The **LIBAD4** features the following triggers:

C

```
#define AD_TRG_NONE
#define AD_TRG_POSITIVE
#define AD_TRG_NEGATIVE
#define AD_TRG_INSIDE
#define AD_TRG_OUTSIDE
#define AD_TRG_NEVER
```

A trigger can be set separately for each channel. The individual trigger conditions are linked with **`or`**, i.e. the first channel meeting the trigger condition sets off the trigger of the DAQ system.

The element **`trg_mode`** should be **`AD_TRG_NONE`** for all channels which are not supposed to trigger. If all channels of a scan are set to **`AD_TRG_NONE`**, the scan will be carried out without trigger, i.e. the recorded values are stored right away.

If all channels of a scan are set to **`AD_TRG_NEVER`**, no trigger is set off at all. In this case, the scan runs until the function **`ad_stop_scan()`** is explicitly called.

The trigger conditions **`AD_TRG_POSITIVE`** (Positive Edge) and **`AD_TRG_NEGATIVE`** (Negative Edge) set off a trigger as soon as a sample overruns or underruns a certain value defined by "Trigger level 1" (**`struct ad_scan_cha_desc,`** parameter **`trg_par[0]`**). If operating DAQ systems with 12 and 16-Bit resolution, the values for the 16-Bit trigger level must be assigned to the lower 16-Bit of the trigger level parameter. A "Positive Edge" trigger, for example, requires that the channel values must first be below the trigger level before exceeding the level sets off the trigger.

The trigger conditions AD_TRG_INSIDE and AD_TRG_OUTSIDE set off a trigger as soon as a sample is within or outside a certain range defined by "Trigger level 1" (**`struct ad_scan_cha_desc`**, parameter **`trg_par[0]`** for minimum) and "Trigger level 2" (**`struct ad_scan_cha_desc`** parameter **`trg_par[1]`** for maximum). In contrast to an edge trigger, only the current sample is decisive to set off a window trigger.

5.2.2 struct ad_scan_desc

The global settings of a scan procedure are specified in the **`struct ad_scan_desc`** structure looking like that:

© BMC Messsysteme GmbH

C

```
struct ad_scan_desc
{
  double sample_rate;
  ...
  uint64_t prehist;
  uint64_t posthist;
  uint32_t ticks_per_run;
  uint32_t bytes_per_run;
  uint32_t samples_per_run;
  uint32_t flags;
  ...
};
```

The elements of the structure bear the following meaning:

`sample_rate`

Determines the sampling rate of the scan (in seconds). To reach 100Hz sampling rate, for example, the value 0.01 must be used.

`prehist`

Sets the length of the prehistory (only if trigger is used, otherwise set to 0).

`posthist`

Sets the length of the posthistory.

`ticks_per_run`

Is required for continuous scans specifying the size of the blocks used to get the measuring values from the DAQ system. In **`ticks_per_run`** the LIBAD4 then returns the block size used in the buffer to send the sampled values from the device.

`bytes_per_run`

Is returned by the **LIBAD4** specifying the buffer size for **`ad_get_next_run()`** (in bytes) if the internal memory management of the measuring values has not been activated.

`samples_per_run`

Is provided by the **LIBAD4** specifying the number of measuring values of a buffer returned by the **`ad_get_next_run_f()`** function if the internal memory management of the measuring values has not been activated.

`flags`

The **`AD_SF_SAMPLES`** bit in **`flags`** defines the memory management of the measuring data. If the **`AD_SF_SAMPLES`** bit is set, internal memory management of the measuring values will be activated.. If the **`AD_SF_SAMPLES`** bit is not set, an individual memory management must be realized.

- **Elements of the structure which are not used or documented must necessarily be set to 0!**
- **The internal memory management of measuring values can only be used for DAQ systems which scan and store independantly (e.g. LAN-AD16fx / AMS42/84-LAN16fx, USB-AD16f / AMS42/84-USB, USB-AD14f, PCIe-BASE / PCI-BASEII / PCI-PIO with MAD/MADDA modules).**
- **If the internal memory management of the measuring values has been activated, the routine `ad_poll_scan_state()` must continuously be called. Reading out measuring values from the internal memory is done with the routines `ad_get_samples()`, `ad_get_samples_f()`, or `ad_get_samples_f64()`.**

© BMC Messsysteme GmbH

5.2.3 struct ad_scan_state

During a running scan, the **LIBAD4** returns the scan state in the **`struct ad_scan_state`** structure:

C

```
struct ad_scan_state
{
  int32_t flags;
  int32_t runs_pending;
  int64_t posthist;
};
```

The elements of the structure bear the following meaning:

`flags`

Shows the scan state (see below).

`posthist`

Contains the number of measuring values after triggering. If no trigger is set, the number of currently sampled measuring values will be passed.

`runs_pending`

Shows if the next RUN is ready to be read out. If this flag is not zero, the next RUN can be read out with **`ad_get_next_run()`**.

The scan state is passed by the **`flags`** element. This element can be used to find out if the trigger has already been set off and if the scan is still running:

C

```
struct ad_scan_state state;

...

if (state & AD_SF_TRIGGER)
  /* scan has triggered */

...

if (state & AD_SF_SCANNING)
  /* scan is still running */
```

The **`struct ad_scan_state`** structure can be requested by the **LIBAD4** either when reading out the measuring values with **`ad_get_next_run()`** or by explicitly calling **`ad_poll_scan_state()`**.

5.2.4 struct ad_scan_pos

C

```
struct ad_scan_pos
{
  uint32_t run;
  uint32_t offset;
};
```

The structure provides information about the individual scan runs.

The elements of the structure bear the following meaning:

RUN number of the scan

Offset used in the respective RUN of the scan

5.2.5 struct ad_cha_layout

C

```
struct ad_cha_layout
{
  struct ad_scan_pos start;
  int64_t prehist_samples;
  int64_t posthist_samples;
  double t0;
};
```

The elements of the structure bear the following meaning:

Position of the first scanned measuring value

Number of measuring values before triggering

Number of measuring values after triggering

Time in seconds until the trigger event occurs

5.3 Memory-only Scan

A "memory-only" scan is started and run by calling the three functions **`ad_start_mem_scan()`**, **`ad_get_next_run()`** and **`ad_stop_scan()`**. All recorded samples of such a scan are stored in the (physically existing) main memory of the PC.

The example code in the following chapter demonstrates how to start a scan and read out the recorded values.

5.3.1 Starten eines Scans

To be able to start the **`ad_start_mem_scan()`** function, the channels to be sampled must have been defined first. The following example generates the channel description for two channels (analog input 1 and analog input 3). Both channels are stored 1:1.

© BMC Messsysteme GmbH

C

```c
struct ad_scan_cha_desc chav[2];
...
memset (chav, 0, sizeof(chav));

chav[0].cha = AD_CHA_TYPE_ANALOG_IN|1;
chav[0].store = AD_STORE_DISCRETE;
chav[0].ratio = 1;
chav[0].trg_mode = AD_TRG_NONE;

chav[1].cha = AD_CHA_TYPE_ANALOG_IN|3;
chav[1].store = AD_STORE_DISCRETE;
chav[1].ratio = 1;
chav[1].trg_mode = AD_TRG_NONE;
```

Besides that, the global scan parameters must be set in the **`struct ad_scan_desc`** structure. The following example sets the sampling rate to 1kHz and stores 500 sampled values (per channel).

C

```c
struct ad_scan_desc sd;
...
memset (&sd, 0, sizeof(sd));

sd.sample_rate = 0.001f;
sd.prehist = 0;
sd.posthist = 500;
```

Afterwards **`ad_start_mem_scan()`** can be called:

C

```c
int32_t rc;
...
rc = ad_start_mem_scan (adh, &sd, 2, chav);
if (rc != 0)
  return rc;
...
```

Now the scan process is running in the background and is terminated after 0.5sec (500x 1msec).

5.3.2 Reading out Measuring Values

Recorded values are read out by calling the **`ad_get_next_run()`** or **`ad_get_next_run_f()`** function. Compared to the **`ad_get_next_run ()`** function returning the samples directly from the DAQ system (as 16-bit values), the **`ad_get_next_run_f ()`** function passes float values, which are (depending on the measuring range) already converted into the relating voltage values. In case of a "memory-only" scan, both functions are disabled until all measuring values have been stored (i.e. for 0.5 seconds in this case).

Both functions expect a pointer to a data buffer, which must be big enough to store the whole amount of measuring values. The memory will be overwritten otherwise and the program will crash!

The minimum size of the buffer for **ad_get_next_run()** can be determined with the **bytes_per_run** element of the **struct ad_scan_desc** structure. A buffer to be filled by **ad_get_next_run_f ()** must provide storage for **samples_per_run** float values at least.

In this case, 2 channels with 500 measuring values each are stored so that the buffer must feature a size of 1000 float values at least:

C

```
float samples[1000];
...

ASSERT (sd.samples_per_run <= 1000);

rc = ad_get_next_run_f (adh, NULL, NULL, samples);

...
```

After successfully calling the function, the array **samples[]** is filled with the following measuring values (the samples a_i are provided by analog input 1, the samples b_i by analog input 3):

Array index	0	1	2	…	498	499	500	501	502	…	998	999	...
Time (in msec)	**0**	**1**	**2**	**…**	**498**	**499**	**0**	**1**	**2**	**…**	**498**	**499**	...
Sample	a_1	a_2	a_3	**…**	a_{499}	a_{500}	b_1	b_2	b_3	**…**	b_{499}	b_{500}	...

5.3.3 Stopping a Scan

If a scan process has been started successfully (return value of **ad_start_scan()** was 0), it must be stopped with **ad_stop_scan**.

The scan must also be stopped if an error has been returned upon reading out measuring values. As long as the scan has not been stopped, a new scan cannot be started.

© BMC Messsysteme GmbH

The following example code stops the scan:

C

```c
int32_t scan_result;
...

rc = ad_stop_scan (adh, &scan_result);

...
```

5.4 Continuous Scan

Besides the "memory-only" scan, the **LIBAD4** features the continuous scan. In this case, the measuring values are passed to the caller block by block enabling him to analyze the measuring values during the scan and to make adjustments if necessary.

The measuring values are grouped in "RUNs", which are passed to the caller by the **LIBAD4**. The number of measuring values belonging to a RUN can be defined by the caller with the **`ticks_per_run`** element of the **`struct ad_scan_desc`** structure.

This parameter can also take extreme values. If **`ticks_per_run`** is set to 1, for example, the **LIBAD4** generates one RUN per measured value. On the other hand, this configuration only allows very small sampling rates, of course.

It is the caller's responsibility to set the number of samples per RUN so that **`ad_get_next_run()`** can be called often enough to prevent an overrun of the samples. Otherwise, the scan will be aborted by the **LIBAD4**.

5.4.1 Composition of a RUN

The number of samples of a RUN is passed to the **LIBAD4** by the **`ticks_per_run`** element of the **`struct ad_scan_desc`** structure. The following example splits the recorded values of the scan into two RUNs with 250 samples each (per signal).

As this example shows, a continuous scan is started with **`ad_start_scan ()`** (unlike **`ad_start_mem_scan()`**). In this case, the array **`ticks_per_run`** of the **`struct ad_scan_desc`** structure must be have been defined before.

The example generates the following two RUNs during the scan, the first RUN being returned by **`ad_get_next_run()`** 250msec after scan start, the second 500msec after scan start.

C

```
int32_t rc;
struct ad_scan_cha_desc chav[2];
struct ad_

scan_desc sd;

...

memset (&chav, 0, sizeof(chav));
memset (&sd, 0, sizeof(sd));

chav[0].cha = AD_CHA_TYPE_ANALOG_IN|1;
chav[0].store = AD_STORE_DISCRETE;
chav[0].ratio = 1;
chav[0].trg_mode = AD_TRG_NONE;

chav[1].cha = AD_CHA_TYPE_ANALOG_IN|3;
chav[1].store = AD_STORE_DISCRETE;
chav[1].ratio = 1;
chav[1].trg_mode = AD_TRG_NONE;

sd.sample_rate = 0.001f;
sd.prehist = 0;
sd.posthist = 500;
sd.ticks_per_run = 250;

rc = ad_start_scan (adh, &sd, 2, chav);
if (rc != 0)
  return rc;
...

rc = ad_stop_scan (adh, &scan_result);
...
```

Array index	0	1	2	...	48	49	50	51	52	...	98	99	...
Time (in msec)	**0**	**1**	**2**	**...**	**248**	**249**	**0**	**1**	**2**	**...**	**248**	**249**	...
Sample	a_1	a_2	a_3	**...**	a_{249}	a_{250}	b_1	b_2	b_3	**...**	b_{249}	b_{250}	...

RUN #0

Array index	0	1	2	...	248	249	250	251	252	...	498	499	...
Time (in msec)	**250**	**251**	**252**	**...**	**498**	**499**	**250**	**251**	**252**	**...**	**498**	**499**	...
Sample	a_{251}	a_{252}	a_{253}	**...**	a_{499}	a_{500}	b_{251}	b_{252}	b_{253}	**...**	b_{499}	b_{500}	...

RUN #1

© BMC Messsysteme GmbH

The following example code reads out the RUNs during a scan:

C

```c
struct ad_scan_state state;
uint8_t *data, *p;
uint32_t samples, runs, run_id;
int32_t rc;
...

/* alloc enough space to hold all those runs */
samples = sd.prehist + sd.posthist;
runs = (samples + sd.ticks_per_run-1) / sd.ticks_per_run;
data = malloc (runs * sd.bytes_per_run);
if (data == NULL)
  /* error handling ... */

p = data;
state.flags = AD_SF_SCANNING;

while (state.flags & AD_SF_SCANNING)
  {
    rc = ad_get_next_run (adh, &state, &run_id, p);
    if (rc != 0)
      /* error handling ... */

    printf ("got run %d (%d pending)\n",
            run_id, state.runs_pending);

    p += sd.bytes_per_run;
  }

rc = ad_stop_scan (adh, &scan_result);
...
```

5.4.2 One Sample per RUN

C

```c
struct ad_scan_cha_desc chav[2];
struct ad_scan_desc sd;
int32_t rc;

...

memset (&chav, 0, sizeof(chav));
memset (&sd, 0, sizeof(sd));

chav[0].cha = AD_CHA_TYPE_ANALOG_IN|1;
chav[0].store = AD_STORE_DISCRETE;
chav[0].ratio = 1;
chav[0].trg_mode = AD_TRG_NONE;

chav[1].cha = AD_CHA_TYPE_ANALOG_IN|3;
chav[1].store = AD_STORE_DISCRETE;
chav[1].ratio = 1;
chav[1].trg_mode = AD_TRG_NONE;

sd.sample_rate = 0.010f;
sd.prehist = 0;
sd.posthist = 500;
sd.ticks_per_run = 1;

rc = ad_start_scan (adh, &sd, 2, chav);
if (rc != 0)
  return rc;

...
```

If **`ticks_per_run`** is set to **`1`**, RUNs with one recorded value per signal are created:

The example above generates 500 RUNs with the following content:

Array index	0	1
Time	0	0
Sample	a_1	b_1

RUN #0

Array index	0	1
Time	10	10
Sample	a_2	b_2

RUN #1

Array index	0	1
Time	4980	4980
Sample	a_{499}	b_{499}

RUN #498

© BMC Messsysteme GmbH

Array index	0	1
Time	4990	4990
Sample	a_{500}	b_{500}

RUN #499

5.4.3 Signals with Different Storage Ratio

The two previous examples (see "One Sample per RUN", p. 33) describe the structure of a run for signals stored with 1:1 ratio. This chapter shows an example using 1:5 storage ratio.

If a signal is stored with a ratio other than 1:1, a difference must me made between sample rate and storage ratio. The sample rate is defined for all channels of the DAQ system by the **`sample_rate`** element of the **`struct ad_scan_desc`** structure. The storage ratio can differ from channel to channel. It results of the parameter **`ratio`** of the **`struct ad_scan_desc`** structure by dividing the storage ratio by **`ratio`**.

The following diagram shows a scan of two inputs with 2ms (50Hz) sampling rate. Input **`a`** is stored 1:1, input **`b`** saves the mean value of 5 samples.

Time (in msec)	0	2	4	6	8	10	12	14	16	18	20	22	24	...
Sample Input a	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	...
Sample Input b	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	...
Stored value Input a	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	...
Stored value Input b					$\frac{1}{5}\sum b_i$					$\frac{1}{5}\sum b_i$				

After at least one sample per channel has been stored for each RUN, the different storage ratios determine the minimum size of a RUN.

Here the smallest possible RUN consists of five sampling pulses (**`ticks_per_run`** = 5) containing five samples of input **`a`** and the mean value of the five samples of input **`b`**:

Array index	0	1	2	3	4
Time (in mecs)	0	2	4	6	8
Input a	a_1	a_2	a_3	a_4	a_5
Input b					$\frac{1}{5}\sum b_i$

If several sampling pulses are combined to a RUN, the stored values per signal are successively arranged (example for **`ticks_per_run`** = 250):

Feldindex	0	1	2	...	248	249	250	251	252	...	398	399

Zeit (in ms)	0	2	4	...	496	498	8	18	28	...	488	498
Werte	a_1	a_2	a_3	...	a_{249}	a_{250}	$\frac{1}{5}\sum b_i$	$\frac{1}{5}\sum b_i$	$\frac{1}{5}\sum b_i$	...	$\frac{1}{5}\sum b_i$	$\frac{1}{5}\sum b_i$

5.5 Scan with Triggering

The **LIBAD4** features the possibility to scan with a trigger. In this case, the internal memory management of the measuring values must be activated (**`AD_SF_SAMPLES`** bit of the **`flags`** element of the **`struct ad_scan_desc`** structure is set).

The number of measuring values before triggering (prehistory) and the number of measuring values after triggering (posthistory) is freely adjustable for the scan with the **`struct ad_scan_desc`** structure.

Besides that, a trigger condition can be defined for each channel with the **`struct ad_scan_cha_desc`** structure. If one of the trigger conditions applies, the trigger is set off and the scan is finished after the posthistory has expired.

Continuous reading of measuring values is possible with the command **`ad_poll_scan_state()`** also polling the current scan state. As long as the **`AD_SF_SCANNING`** bit is set in the **`flags`** element of the **`struct ad_scan_state`** structure, the scan is running. As soon as the bit **`AD_SF_TRIGGER`** is set, the scan has triggered, i.e. at least one of the trigger conditions has been achieved.

Reading out measuring values is done with the routines **`ad_get_samples()`**, **`ad_get_samples_f()`**, or **`ad_get_samples_f64()`**. The routine **`ad_get_samples()`** returns the measuring values directly from the DAQ system**`,`** **`ad_get_samples_f ()`** or **`ad_get_samples_f64 ()`** pass float or double values, which are (depending on the measuring range) already converted into the corresponding voltage values. With these routines, data of one scan channel in specific can be read out. Number and start position of the data to be read out are passed when calling the function. To get information about the data memory of one measuring channel **`ad_get_sample_layout()`** is used.

The installation of the Libad4 SDK under Windows® contains a C/C++ to program a scan with triggering.

5.5.1 Parameters for scan channels with triggering

© BMC Messsysteme GmbH

C

```c
  uint32_t data;
  int rng;
  struct ad_scan_cha_desc chav[2];

  rng = 2; /* e.g. +/-5V for usb-ad16f */

  /* need to ensure everything in chav is zero */
  memset (&chav, 0, sizeof(chav));

  chav[0].cha = AD_CHA_TYPE_ANALOG_IN | 1;
  chav[0].store = AD_STORE_DISCRETE;
  chav[0].ratio = 1;
  chav[0].range = rng;

#define TRIGGER_VALUE 2.5

  /* setup trigger for the 1st channel:
   * positive trigger at the defined TRIGGER_VALUE
   */
  chav[0].trg_mode = AD_TRG_POSITIVE;
  rc = ad_float_to_sample(adh, chav[0].cha, rng,
                          TRIGGER_VALUE, &data);
  /* Note: The data have to be placed in the lower 16-bit
   * for all 12-bit and 16-bit devices, e.g. usb-ad16f,
   * usb-ad, mad12a, mad16a, mad16f, im-ad25a, etc.
   */
  printf("Trigger Value %8.3f = hex 0x%04x (rc=%d)\n",
         TRIGGER_VALUE, data>>16, rc);
  chav[0].trg_par[0] = data>>16;
  chav[0].trg_par[1] = 0;
```

C

```c
  /* 2nd scan channel: digital */
  chav[1].cha = AD_CHA_TYPE_DIGITAL_IO | 1;
  chav[1].store = AD_STORE_DISCRETE;
  chav[1].ratio = 1;
  chav[1].range = 0;

  /* set up trigger for the digital channel:
   * trigger when low state at digital line 1.
   * Note:
   * Does NOT trigger on a change in the digital
   * state, it triggers at the digital state condition
   */
  chav[1].trg_mode = AD_TRG_DIGITAL;
  /* trigger condition:
   *  (data and trg_par[0] xor trg_par[1] != 0)
   */
  chav[1].trg_par[0] = 0x0001;  /* and mask value */
  chav[1].trg_par[1] = 0x0001;  /* xor mask value */
```

C

```
struct ad_scan_desc sd;

memset (&sd, 0, sizeof(sd));

sd.sample_rate = 0.001f;
sd.prehist = 100;
sd.posthist = 500;
sd.ticks_per_run = 200;

/* Scans with trigger need the internally managed
 * samples memory to be activated!
 */
sd.flags = AD_SF_SAMPLES;
```

C

```
  /* start scan
   */
  rc = ad_start_scan (adh, &sd, CHAC, chav);

  if (rc < 0)
    /* error */
  struct ad_scan_state state;
  state.flags = AD_SF_SCANNING;
  while (state.flags & AD_SF_SCANNING)
    {
      rc = ad_poll_scan_state (adh, &state);
      if (rc != 0)
        /* error */

      if ((state.flags & AD_SF_TRIGGER) == 0)
        ; /* before trigger */
      else
        ; /* after trigger */
    }
```


© BMC Messsysteme GmbH

C

```
    /* Big enough for all data (see above)!
     */
    float tmp[600];
    uint32_t nval;
    struct ad_sample_layout layout;

    for (int i = 0; i < 2; i++)
      {
        /* get information about the scan channel i
         */
        ad_get_sample_layout (adh, 0, &layout);
        nval = 600;
        rc = ad_get_samples_f(adh, i, AD_STORE_DISCRETE,
             layout.start, &nval, tmp);
        printf ("\nchannel #%d", i);
        int j = 0;
        while (j < ((int) nval))
          {
            printf ("%8.3f\n", data[j++]);
          }
      }
```

5.6 Functions Description (Scan)

5.6.1 ad_start_mem_scan

Prototype

```
int32_t
ad_start_mem_scan (int32_t adh,
                   struct ad_scan_desc *scan_desc,
                   uint32_t chac,
                   struct ad_scan_cha_desc *chav);
```

C

```
struct ad_scan_cha_desc chav[2];
struct ad_scan_desc sd;
int32_t rc;
...

memset (&chav, 0, sizeof(chav));
memset (&sd, 0, sizeof(sd));

/* sample and store analog input #1 */
chav[0].cha = AD_CHA_TYPE_ANALOG_IN|1;
chav[0].store = AD_STORE_DISCRETE;
chav[0].ratio = 1;
chav[0].trg_mode = AD_TRG_NONE;

/* sample and store analog input #3 */
chav[1].cha = AD_CHA_TYPE_ANALOG_IN|3;
chav[1].store = AD_STORE_DISCRETE;
chav[1].ratio = 1;
chav[1].trg_mode = AD_TRG_NONE;

/* 1kHz, 500 samples per signal /
sd.sample_rate = 0.001f;
sd.prehist = 0;
sd.posthist = 500;

rc = ad_start_mem_scan (adh, &sd, 2, chav);
if (rc != 0)
  /* error handling */
```

Starts a "memory-only" scan. A pointer to an element of the **struct ad_scan_desc** structure, the number of channels to be scanned and an array of elements of the **struct ad_scan_cha_desc** structure are passed to the function.

Due to restrictions of most of DAQ cards, it is essential to specify the input channels in ascending order in the array chav[]. If counters and digital channels are scanned in addition to analog channels, all the analog channels must be specified first, then all counters and finally the digital channels!

The arrays **sample_rate**, **ticks_per_run**, **bytes_per_run** and **samples_per_run** of the **struct ad_scan_desc** structure are recalculated according to the set parameters (see "**ad_calc_run_size**", p. 43).

© BMC Messsysteme GmbH

5.6.2 ad_start_scan

Prototype

```
int32_t
ad_start_scan (int32_t adh,
               struct ad_scan_desc *scan_desc,
               uint32_t chac,
               struct ad_scan_cha_desc *chav);
```

Unlike **ad_start_mem_scan()**, the **ad_start_scan()** function analyzes the **ticks_per_run** element of the **struct ad_scan_desc** structure so that a scan can be divided into several RUNs (see "Continuous Scan", p. 30).

Due to restrictions of most of DAQ cards, it is essential to specify the input channels in ascending order in the array chav[]. If counters and digital channels are scanned in addition to analog channels, all the analog channels must be specified first, then all counters and finally the digital channels!

The arrays **sample_rate**, **ticks_per_run**, **bytes_per_run** and **samples_per_run** of the **struct ad_scan_desc** structure are recalculated according to the specified parameters (see "**ad_calc_run_size**", p. 43).

5.6.3 ad_get_sample_layout

Prototype

```
int32_t
ad_get_sample_layout (int32_t adh, int32_t index, struct
ad_sample_layout *layout);
```

Returns information about the data memory for the scan channel **index** if the internal memory management of the measuring values has been activated. The scan channel numbering starts with **index** = 0.

The **struct ad_sample_layout** structure consists of:

C

```
struct ad_sample_layout
{
  uint64_t buffer_start;
  uint64_t start;
  uint64_t prehist_samples;
  uint64_t posthist_samples;
  };
```

The elements of the structure bear the following meaning:

buffer_start

Position of the first measuring value in the data memory of the scan

start

Position of the first measuring value of the prehistory in the data memory of the scan

`prehist_samples`

Number of available measuring values in the data memory before triggering. The prehistory ranges from the position **`start`** to (**`start`** + **`prehist_samples`**).

`posthist_samples`

Number of available measuring values after triggering. The posthistory ranges from the position (**`start`** + **`prehist_samples`**) to (**`start`** + **`prehist_samples`** + **`posthist_samples`**).

The internal memory management of measuring values (`AD_SF_SAMPLES` bit of the `flags` element is set in the `struct ad_scan_desc`) must be activated to use this routine.

5.6.4 ad_get_samples

Prototype

```
int32_t
ad_get_samples (int32_t adh, int32_t index, int32_t type, uint64_t
offset, uint32_t *n, void *buf);
```

Returns the measuring values for the scan channel **`index`** as float or double values if the internal memory management of the measuring values has been activated. The scan channel numbering starts with **`index`** = 0.

The measuring values provided by the DAQ system have been converted into the corresponding voltage values depending on the measuring range chosen. Starting from the position **`offset`**, **`n`** measuring values in float format are read out of the data memory of the scan channel **`index`**. The position **`offset`** must not be smaller than **`buffer_start`**, the element of the **`struct ad_get_sample_layout`** structure.

The number of read-out measuring values is returned by the parameter **`n`**. The parameter **`type`** decides which data of the data memory are written to the provided array **`buf`**. Only data types (Discrete, Minimum, Maximum, etc) can be extracted that have been specified when selecting the storage mode (**`store`** element of **`struct ad_scan_desc`**) of the scan channel.

- **The function expects a pointer to a data buffer. It must be big enough to store all measuring values. The memory will be overwritten otherwise and the program will crash!**
- **The internal memory management of measuring values (AD_SF_SAMPLES bit of the flags element is set in the Fehler! Verweisquelle konnte nicht gefunden werden.) must be activated to use this routine.**

5.6.5 ad_get_samples_f

Prototype

```
int32_t
ad_get_samples_f (int32_t adh, int32_t index, int32_t type, uint64_t
offset, uint32_t *n, float *buf);
```


© BMC Messsysteme GmbH

Returns the measuring values for the scan channel **index** as float or double values if the internal memory management of the measuring values has been activated. The scan channel numbering starts with **index** = 0.

The measuring values provided by the DAQ system have been converted into the corresponding voltage values depending on the measuring range chosen. Starting from the position **offset**, **n** measuring values in float format are read out of the data memory of the scan channel **index**. The position **offset** must not be smaller than **buffer_start**, the element of the **struct ad_get_sample_layout** structure.

The number of read-out measuring values is returned by the parameter **n**. The parameter **type** decides which data of the data memory are written to the provided array **buf**. Only data types (Discrete, Minimum, Maximum, etc) can be extracted that have been specified when selecting the storage mode (**store** element of **struct ad_scan_desc**) of the scan channel.

- **The function expects a pointer to a data buffer. It must be big enough to store all measuring values. The memory will be overwritten otherwise and the program will crash!**
- **The internal memory management of measuring values (AD_SF_SAMPLES bit of the flags element is set in the struct ad_scan_desc) must be activated to use this routine.**
- **The routine ad_get_samples_f64() should be used for measuring channels with more than 16-bit memory depth (e.g. 32-bit counter of the USB-OI16 / PCIe-BASE / PCI-BASEII / PCI-PIO).**

5.6.6 ad_get_samples_f64

Prototype

```
int32_t
ad_get_samples_f64 (int32_t adh, int32_t index, int32_t type, uint64_t
offset, uint32_t *n, double *buf);
```

Returns the measuring values for the scan channel **index** as float or double values if the internal memory management of the measuring values has been activated. The scan channel numbering starts with **index** = 0.

The measuring values provided by the DAQ system have been converted into the corresponding voltage values depending on the measuring range chosen. Starting from the position **offset**, **n** measuring values in float format are read out of the data memory of the scan channel **index**. The position **offset** must not be smaller than **buffer_start**, the element of the **struct ad_get_sample_layout** structure.

The number of read-out measuring values is returned by the parameter **n**. The parameter **type** decides which data of the data memory are written to the provided array **buf**. Only data types (Discrete, Minimum, Maximum, etc) can be extracted that have been specified when selecting the storage mode (**store** element of **struct ad_scan_desc**) of the scan channel.

- **The function expects a pointer to a data buffer. It must be big enough to store all measuring values. The memory will be overwritten otherwise and the program will crash!**
- **The internal memory management of measuring values (AD_SF_SAMPLES bit of the flags element is set in the struct ad_scan_desc) must be activated to use this routine.**

5.6.7 ad_calc_run_size

Prototype

```
int32_t
ad_calc_run_size (int32_t adh,
                  struct ad_scan_desc *scan_desc,
                  uint32_t chac,
                  struct ad_scan_cha_desc *chav);
```

Calculates the arrays **sample_rate**, **ticks_per_run**, **bytes_per_run** and **samples_per_run** of the **struct ad_scan_desc** structure according to the specified parameters.

The arrays are calculated as if calling the **ad_start_scan()** function, but without starting the scan process. Like when calling the **ad_start_scan()** function, the calculation or adjustment proceeds as follows:

sample_rate

Is set to the actually possible sampling period (most of the DAQ cards can only set the sampling period in fixed steps).

ticks_per_run

Must be adjusted accordingly so that one value of each signal at least will be stored and/or one single RUN will fit into the internal memory of the driver.

bytes_per_run

Is calculated by the **LIBAD4** providing the number of bytes of the buffer for **ad_get_next_run()**.

samples_per_run

Is calculated by the **LIBAD4** providing the number of float values within a buffer for **ad_get_next_run_f()**.

The buffer size for **ad_get_next_run_f()** can be calculated with **samples_per_run**:

C

```c
struct ad_scan_desc sd;
float *data;
int32_t rc;
...

rc = ad_calc_run_size (adh, &sd, 2, chav);
if (rc != 0)
  return rc;

data = malloc (sd.samples_per_run * sizeof(float));
...
```

5.6.8 ad_get_next_run

Prototype

```
int32_t
ad_get_next_run (int32_t adh,
                 struct ad_scan_state *state,
                 uint32_t *run, void *p);
```

 © BMC Messsysteme GmbH

Returns the measuring values of a scan.

The **`ad_get_next_run()`** function returns the measuring values directly from the DAQ system (i.e. as 16-bit or 32-bit values depending on the memory depth of the measuring channel). The lower range limit relates to the value **`0x0000`**, the upper range limit to the value **`0xffff`** or **`0xffffffff`** (more precisely, the upper limit relates to the value **`0x10000`** or **`0x100000000`**, which will not be reached).

> **The recorded values are returned in "network byte order", i.e. they are not in the byte order of a x86 CPU!**

The function does not return as long as the samples of a RUN are not available. In a "memory-only" scan, this means the function does not return until the end of a scan (because a "memory-only" scan stores all samples in one single RUN).

5.6.9 ad_get_next_run_f

Prototype

```
int32_t
ad_get_next_run_f (int32_t adh,
                   struct ad_scan_state *state,
                   uint32_t *run, float *p);
```

Returns the measuring values of a scan as float values.

The values provided by the DAQ system are (depending on the measuring range) converted into the corresponding voltage values.

The function does not return as long as the measuring values of a RUN are not available. In a "memory-only" scan, this means the function does not return until the end of a scan (because a "memory-only" scan stores all samples in one single RUN).

> **The routine ad_get_next_run_f64() should be used for**
>
> **measuring channels with more than 16-bit memory depth (e.g. 32-**
>
> **bit counter of the Fehler! Verweisquelle konnte nicht gefunden werden. / PCIe-BASE / PCI-BASEII / PCI-PIO).**

5.6.10 ad_get_next_run_f64

Prototype

```
int32_t
ad_get_next_run_f64 (int32_t adh,
                   struct ad_scan_state *state,
                   uint32_t *run, double *p);
```

Returns the measuring values of a scan as float values.

The values provided by the DAQ system are (depending on the measuring range) converted into the corresponding voltage values.

The function blocks until the measuring values of a run have arrived. This means that for a "memory-only scan" the function blocks till the end of the scan (because a "memory-only" scan saves all measuring values in one run).

© BMC Messsysteme GmbH

5.6.11 ad_poll_scan_state

```
Prototype

int32_t
ad_poll_scan_state (int32_t adh,
                    struct ad_scan_state *state);
```

Returns the current scan state like calling the **`ad_get_next_run()`** function. Unlike **`ad_get_next_run()`**, this function does not block.

> **If the internal memory management of the measuring data has be activated (AD_SF_SAMPLES bit of the flags element in the Fehler! Verweisquelle konnte nicht gefunden werden. structure is set) the routine ad_poll_scan_state() must continuously be called.**

5.6.12 ad_stop_scan

```
Prototype

int32_t
ad_stop_scan (int32_t adh, int32_t *scan_result);
```

Finishes the scan. The result of the scan is passed in **`scan_result`** (e.g. an error number if the scan has been aborted because of an overrun).

> **If a scan process has successfully been started (return value of Fehler! Verweisquelle konnte nicht gefunden werden.() was 0), it must be finished with ad_stop_scan ().**

 © BMC Messsysteme GmbH

6 Data Acquisition Systems

6.1 Notes

Input and output channels are identified by channel numbers in the **LIBAD4**. The channel number (32-bit integer) also contains the channel type information distinguishing between analog input, analog output and digital channel. This encoding is integrated in the highest byte of the channel number and must be combined with the channel number itself by the "or" operator (|).

The following channel types are defined in the **LIBAD4**:

C

```
#define AD_CHA_TYPE_ANALOG_IN
#define AD_CHA_TYPE_ANALOG_OUT
#define AD_CHA_TYPE_DIGITAL_IO
#define AD_CHA_TYPE_COUNTER
```

The channel numbers depend on the DAQ system used and are documented in the relating chapters. The first analog input of a USB-AD14f, for example, is defined by the expression **`AD_CHA_TYPE_ANALOG_IN|1`**.

In addition to the channel number, analog channels require information about the measuring range (or output range) used to scan (or to output). Like the channel number, the measuring range depends on the data acquisition system and is documented in the following chapters.

6.2 LAN-AD16fx / AMS42/84-LAN16fx

Open the LAN-AD16fx or AMS42/84-LAN16fx with the **LIBAD4** by passing the string "**`lanbase:<ip-addr>`**" or "**`lanbase:@<sn>`**" to **`ad_open()`**. Here **`<ip-addr>`** must be replaced by the relating IP address or **`<sn>`** by the serial number of the LAN-AD16fx or AMS42/84-LAN16fx. The string "**`lanbase:192.168.1.1`**", for example, opens the LAN device with the IP address 192.168.1.1, and the string "**`lanbase:@157`**" opens the LAN device with the serial number 157.

Opening the device via the serial number is only supported by Windows® and Mac OS X.

DAQ system	Analog	Channel numberr	Measuring range	Output range	Digital	Direction
LAN-AD16fx/ AMS42/84-LAN16fx	16 inputs 2 outputs	1..16 1 .. 2	0 (±1.024V) 1 (±2.048V) 2 (±5.120V) 3 (±10.240V)	0 (±10.24V)	2 ports (16 bit each)	1: port A 2: port B

6.2.1 Channel Numbers LAN-AD16fx / AMS42/84-LAN16fx

The 16 analog inputs of a LAN-AD16fx or AMS42/84-LAN16fx are addressed via the channel numbers 1-16. The 2 analog outputs are reached via channel number 1 and 2. The 16 analog inputs are defined by the following constants:

C

```
#define AI1    (AD_CHA_TYPE_ANALOG_IN|0x0001)
#define AI2    (AD_CHA_TYPE_ANALOG_IN|0x0002)
...
#define AI16   (AD_CHA_TYPE_ANALOG_IN|0x0010)
```

The two analog output channels of a LAN-AD16fx and AMS42/84-LAN16fx are addressed by the following constants:

C

```
#define AO1    (AD_CHA_TYPE_ANALOG_OUT|0x0001)
#define AO2    (AD_CHA_TYPE_ANALOG_OUT|0x0002)
```

The LAN-AD16fx or AMS42/84-LAN16fx provides two 16-bit digital ports. The digital ports are bidirectional and are configured in groups of 8 (see "**`ad_set_line_direction`**", p. **Fehler! Textmarke nicht definiert.**). After boot-up, the first port is set to input, the second to output. The following constants result:

C

```
#define DIO1   (AD_CHA_TYPE_DIGITAL_IO|0x0001)
#define DIO2   (AD_CHA_TYPE_DIGITAL_IO|0x0002)
```

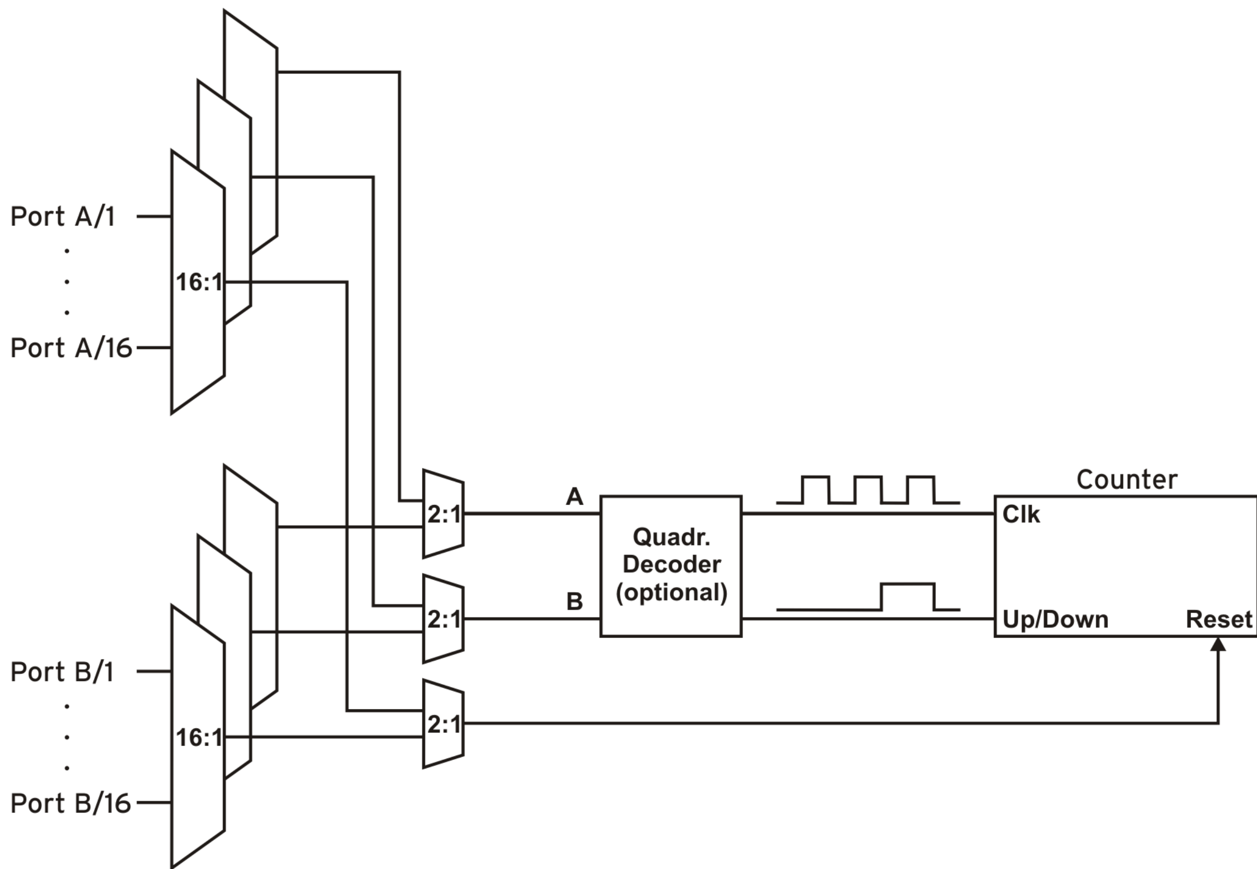
Besides that, the LAN-AD16fx or AMS42/84-LAN16fx features three 32-bit counter inputs. They can be used in different operating modes and must be configured via software before use (see"Configuration of the LAN-AD16fx / AMS42/84-LAN16fx Counters", p. 48. Therefore connect the inputs of the counter (Signal A, Signal B, Reset) to the first three digital input lines of the LAN-AD16fx or AMS42/84-LAN16fx digital port. The following constants are defined for the 32-bit counter input:

C

```
#define CNT1    (AD_CHA_TYPE_COUNTER|0x0001)
#define CNT2    (AD_CHA_TYPE_COUNTER|0x0002)
#define CNT3    (AD_CHA_TYPE_COUNTER|0x0003)
```


© BMC Messsysteme GmbH

6.2.2 Configuration of the LAN-AD16fx / AMS42/84-LAN16fx Counters



For counter settings, the configuration parameters are entered in the **`struct ad_counter_mode`** structure and passed to **`ad_ioctl()`**.

The following example demonstrates the general procedure: It configures the first counter of the LAN-AD16fx and AMS42/84-LAN16fx in the "Counter" operating mode and connects counter input A with the second input pin of the first digital port.

Prototype

```
int32_t
ad_ioctl (int32_t adh, int32_t ioc,
          void *par, int32_t size);
```

C

```
#include "libad.h"
...

struct ad_counter_mode par;
int32_t adh;
int32_t st;

...

  adh = ad_open ("pcibase");

  memset (&par, 0, sizeof(par));
  par.cha = AD_CHA_TYPE_COUNTER|1;
  par.mode = AD_CNT_COUNTER;
  par.mux_a = 1;
  st = ad_ioctl (adh, AD_SET_COUNTER_MODE,
                 &par, sizeof(par));
...
  ad_close (adh);
```

The following source code shows the layout of the **`struct ad_counter_mode`** structure:

C

```
struct ad_counter_mode
{
  uint32_t cha;

  uint8_t mode;
  uint8_t mux_a;
  uint8_t mux_b;
  uint8_t mux_rst;
  uint16_t flags;
  ...
};
```

The elements of the structure bear the following meaning:

`cha`

Determines the counter channel to be configured.

`mode`

Sets the operating mode of the counter.

Operating Mode	BDescription
AD_CNT_COUNTER	The counter channel is used as a simple counter. Input A of the counter is used only. Each positive edge at the input increases the counter..
AD_CNT_UPDOWN	The counter channel is used as an Up/Down counter, i.e. the counter is bidirectional. Input A of the counter is for the pulse input, input B for changing the direction. If input B of the counter is low, each positive edge at input A increases the counter. Otherwise, the positive edge reduces the counter.
AD_CNT_QUAD_DECODER	The counter decodes the two tracks of an incremental encoder. In this case, each edge of the two tracks is decoded.

© BMC Messsysteme GmbH

AD_CNT_PULSE_TIME	Configures the counter for pulse time measurement. In this case, the counter input is connected with an internal clock (60MHz) and will be started and stopped at each edge of input A.

`mux_a, mux_b, mux_rst`

Defines the pins of the two digital ports that are connected to the inputs of the counter. It is not possible to connect the counter inputs with different digital ports (i.e. inputs A, B and *Reset* must either all be connected with pins of port A or all with pins of port B).

`mux_a`, `mux_b` or `mux_rst`	Port/Line
0	PA/1
1	PA/2
2	PA/3
3	PA/4
4	PA/5
5	PA/6
6	PA/7
7	PA/8
8	PA/9
9	PA/10
10	PA/11
11	PA/12
12	PA/13
13	PA/14
14	PA/15
15	PA/16

`mux_a`, `mux_b` or `mux_rst`	Port/Line
16	PB/1
17	PB/2
18	PB/3
19	PB/4
20	PB/5
21	PB/6
22	PB/7
23	PB/8
24	PB/9
25	PB/10
26	PB/11
27	PB/12
28	PB/13
29	PB/14
30	PB/15
31	PB/16

`flags`

Determines the operation mode of the counter inputs. The operation modes can be combined with **`OR`**: e.g. **`AD_CNT_INV_RST|AD_CNT_ENABLE_RST`**.

Operating Mode	Description
AD_CNT_INV_A	Counter input A reacts inversely.
AD_CNT_INV_B	Counter input B reacts inversely.
AD_CNT_INV_RST	Reset input reacts inversely.
AD_CNT_ENABLE_RST	Reset input is activated.

6.3 PCIe-BASE / PCI-BASEII / PCI-PIO

To open the PCIe-BASE, PCI-BASEII or PCI-PIO with the **LIBAD4**, the string "**pcibase**" (or "**pci300**") must be passed to **ad_open()**. When opening the driver, no difference is made between different versions of the PCI(e) data acquisition card.

To distinguish between several cards, the card number is explicitly used (1st card with "**pcibase:0**", 2nd card with "**pcibase:1**", etc.).

A DAQ card is also directly accessible via its serial number. The card with the serial number 157 can be addressed with "**pcibase:@157**", for example.

6.3.1 Digital Ports and Counters

The PCIe-BASE / PCI-BASEII / PCI-PIO provide two 16-bit digital ports.

The counters of the PCI(e)-BASE cards can be used in different operating modes. They must be configured via software before use.

Each input of the counter can be connected with any of the 16 digital inputs of the two digital port. These settings have to be configured also before using the counter (see "Configuration of the Counters", p. 52).

6.3.1.1 PCIe-BASE / PCI-BASEII / PCI-PIO

The digital ports of the PCIe-BASE / PCI-BASEII / PCI-PIO are bidirectional and are configured in groups of 8 (see "**ad_set_line_direction**", p. 20). After boot-up, the first port is set to input, the second to output. The following numbering is used:

C

```c
#define DIO1  (AD_CHA_TYPE_DIGITAL_IO|0x0001)
#define DIO2  (AD_CHA_TYPE_DIGITAL_IO|0x0002)
```

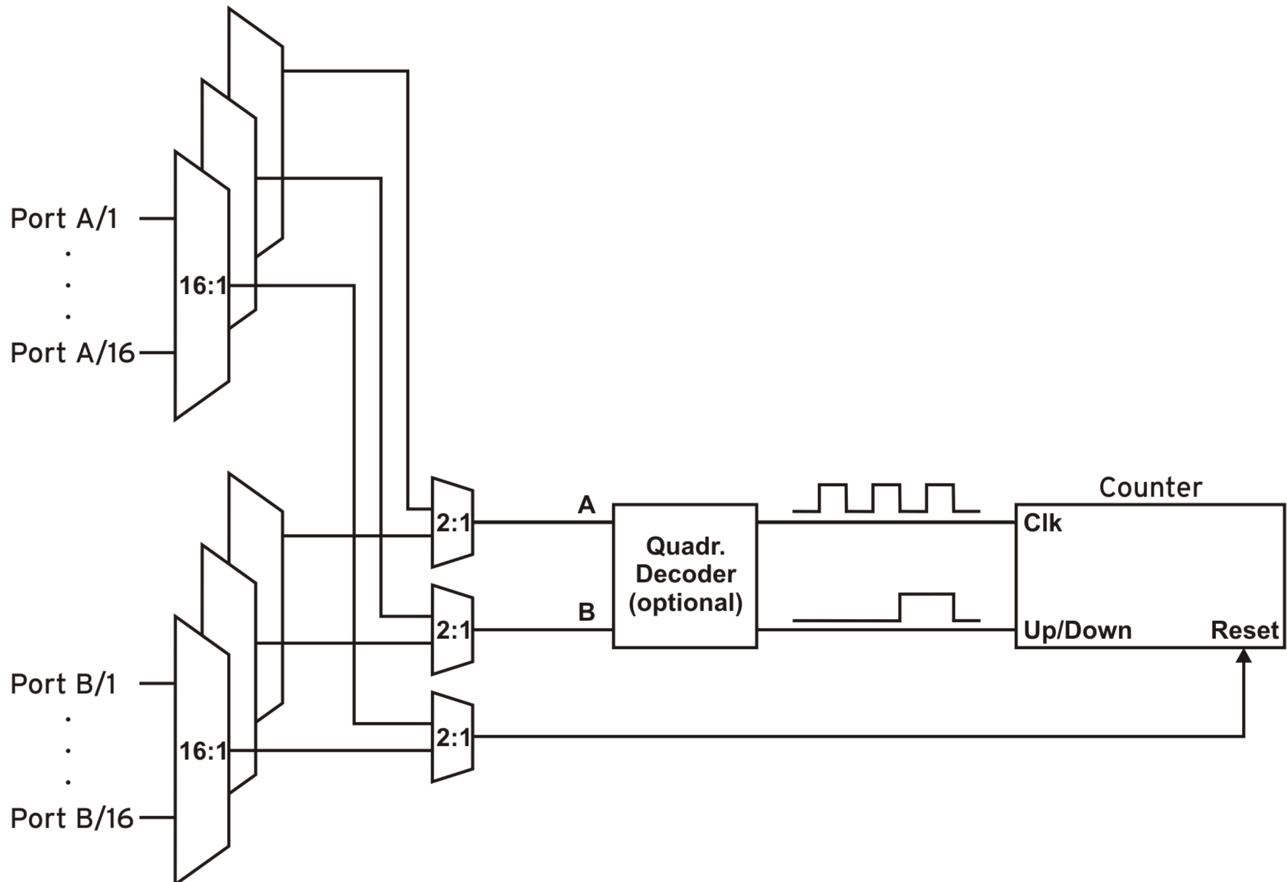
Besides that, the PCIe-BASE / PCI-BASEII, and the PCI-PIO provide three 32-bit counter inputs:

C

```c
#define CNT1   (AD_CHA_TYPE_COUNTER|0x0001)
#define CNT2   (AD_CHA_TYPE_COUNTER|0x0002)
#define CNT3   (AD_CHA_TYPE_COUNTER|0x0003)
```


© BMC Messsysteme GmbH

6.3.1.2 Configuration of the Counters



For counter settings, the configuration parameters are entered in the **`struct ad_counter_mode`** structure and passed to **`ad_ioctl()`**.

The following example demonstrates the general procedure: It configures the first counter of the PCIe-BASE / PCI-BASEII / PCI-PIO in the "Counter" operating mode and connects counter input A with the second input pin of the first digital port.

Prototype

```
int32_t
ad_ioctl (int32_t adh, int32_t ioc,
          void *par, int32_t size);
```

C

```
#include "libad.h"
...

struct ad_counter_mode par;
int32_t adh;
int32_t st;

...

  adh = ad_open ("pcibase");

  memset (&par, 0, sizeof(par));
  par.cha = AD_CHA_TYPE_COUNTER|1;
  par.mode = AD_CNT_COUNTER;
  par.mux_a = 1;
  st = ad_ioctl (adh, AD_SET_COUNTER_MODE,
                 &par, sizeof(par));
...
  ad_close (adh);
```

The following source code shows the layout of the **`struct ad_counter_mode`** structure:

C

```
struct ad_counter_mode
{
  uint32_t cha;

  uint8_t mode;
  uint8_t mux_a;
  uint8_t mux_b;
  uint8_t mux_rst;
  uint16_t flags;
  ...
};
```

The elements of the structure bear the following meaning:

`cha`

Determines the counter channel to be configured.

`mode`

Sets the operating mode of the counter.

Operating Mode	Description
AD_CNT_COUNTER	The counter channel is used as a simple counter. Input A of the counter is used only. Each positive edge at the input increases the counter.
AD_CNT_UPDOWN	The counter channel is used as an Up/Down counter, i.e. the counter is bidirectional. Input A of the counter is for the pulse input, input B for changing the direction. If input B of the counter is low, each positive edge at input A increases the counter. Otherwise, the positive edge reduces the counter.

© BMC Messsysteme GmbH

AD_CNT_QUAD_DECODER	The counter decodes the two tracks of an incremental encoder. In this case, each edge of the two tracks is decoded.

`mux_a, mux_b, mux_rst`

Defines the pins of the two digital ports that are connected to the inputs of the counter. It is not possible to connect the counter inputs with different digital ports (i.e. inputs A, B and *Reset* must either all be connected with pins of port A or all with pins of port B).

mux_a, mux_b or mux_rst	Port/Line
0	PA/1
1	PA/2
2	PA/3
3	PA/4
4	PA/5
5	PA/6
6	PA/7
7	PA/8
8	PA/9
9	PA/10
10	PA/11
11	PA/12
12	PA/13
13	PA/14
14	PA/15
15	PA/16

mux_a, mux_b or mux_rst	Port/Line
16	PB/1
17	PB/2
18	PB/3
19	PB/4
20	PB/5
21	PB/6
22	PB/7
23	PB/8
24	PB/9
25	PB/10
26	PB/11
27	PB/12
28	PB/13
29	PB/14
30	PB/15
31	PB/16

`flags`

Determines the operation mode of the counter inputs. The operation modes can be combined with **`OR:`** e.g. **`AD_CNT_INV_RST|AD_CNT_ENABLE_RST`**.

Operating Mode	Description
AD_CNT_INV_A	Counter input A reacts inversely.
AD_CNT_INV_B	Counter input B reacts inversely.
AD_CNT_INV_RST	Reset input reacts inversely.
AD_CNT_ENABLE_RST	Reset input is activated.

6.3.2 Plug-on Modules

Up to two plug-on modules can be installed on the PCIe-BASE / PCI-BASEII / PCI-PIO. These modules provide additional channels and are described in the following chapters.

6.3.2.1 MADDA16/16n

The first analog input channel of a MADDA16/16n starts with 1. If there is a second analog input module on the PCI(e) multi-function card (not: PCI-PIO), the first input of the second module is addressed via the number 257 (0x100+1).

The module slot on the DAQ card is not relevant. Only the module address determines the assignment of the channels. For example, the MADDA module with the lower address is assigned to the channels 1-16 (analog inputs, channel numbers 0x001 to 0x010) or 1 to 2 (analog outputs, channel numbers 0x001 to 0x002), the MADDA module with the higher address to channel 17-32 (analog inputs, channel numbers 0x101 to 0x110) or 3 to 4 (analog outputs, channel numbers 0x003 to 0x004).

Module	Analog	Channel nubmer	Measuring range	Output range
MADDA16, MADDA16n	16 inputs 2 outputs	1..16 1..2	0 (±1.024V) 1 (±2.048V) 2 (±5.120V) 3 (±10.240V)	0 (±10.24V)

The first 32 analog inputs are defined by the following constants:

C

```
#define AI1   (AD_CHA_TYPE_ANALOG_IN|0x0001)
#define AI2   (AD_CHA_TYPE_ANALOG_IN|0x0002)
...
#define AI16  (AD_CHA_TYPE_ANALOG_IN|0x0010)

/* chas 17 to 32 only if second MADDA module present */
#define AI17  (AD_CHA_TYPE_ANALOG_IN|0x0101)
#define AI18  (AD_CHA_TYPE_ANALOG_IN|0x0102)
...
#define AI32  (AD_CHA_TYPE_ANALOG_IN|0x0110)
```

With two MADDA modules on the DAQ card, the following channel numbers are assigned to the two analog outputs per MADDAmodule:

C

```
#define AO1   (AD_CHA_TYPE_ANALOG_OUT|0x0001)
#define AO2   (AD_CHA_TYPE_ANALOG_OUT|0x0002)

/* chas 3 to 4 only if second MADDA module present */
#define AO3   (AD_CHA_TYPE_ANALOG_IN|0x0101)
#define AO4   (AD_CHA_TYPE_ANALOG_IN|0x0102)
```

6.3.2.2 MDA16-4i/-8i

The channels of a second analog output module are accessible from number 257 (0x100+1) on.

The order of the modules is only defined by the module address and not by the slot on the carrier board so that the channels of the module with the higher address start at 0x101.

 © BMC Messsysteme GmbH

Module	Analog	Channel number	Output range	Range
MDA16-4i	4 outputs	1..4	±10.24V	0
MDA16-8i	8 outputs	1..8	±10.24V	0

The definition of the channel numbers depends on the combination of the output modules on the DAQ card. For example, the following channel numbers are assigned if using an MDA16-2i and an MDA16-4i output module:

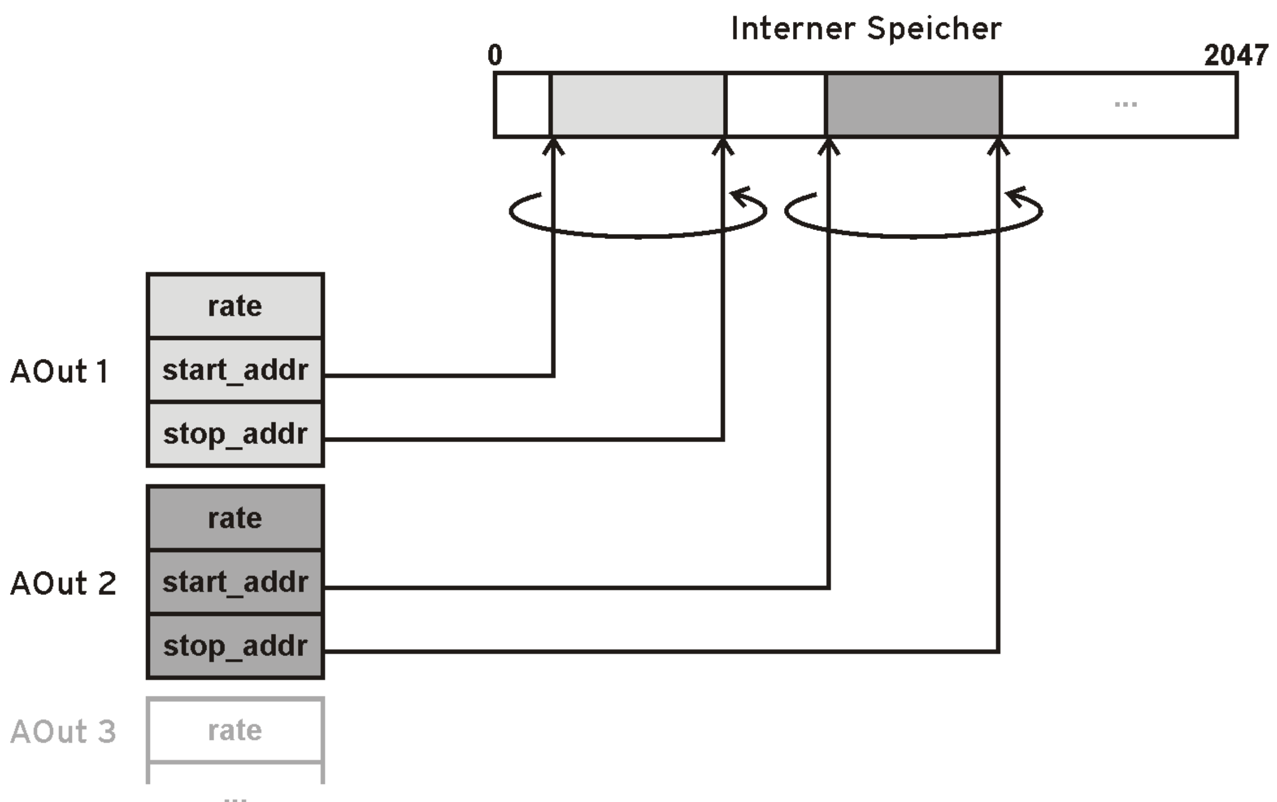
C

```
/* for example a PCI-BASEII with an MDA16-2i (module
 * address 2) and an MDA16-4i (address 3) */

/* MDA16-2i with module address 2  (2 chas) /
#define AO1   (AD_CHA_TYPE_ANALOG_OUT|0x0001)
#define AO2   (AD_CHA_TYPE_ANALOG_OUT|0x0002)

/* MDA16-4i with module address 3 (4 chas)
#define AO3   (AD_CHA_TYPE_ANALOG_OUT|0x0101)
#define AO4   (AD_CHA_TYPE_ANALOG_OUT|0x0102)
#define AO5   (AD_CHA_TYPE_ANALOG_OUT|0x0103)
#define AO6   (AD_CHA_TYPE_ANALOG_OUT|0x0104)
```

6.3.2.3 Functionsgenerator of the MDA16-4i/-8i



The module provides a memory for 2048 measuring values, in which any signal forms can be load. Each output channel of the MDA16-4i/-8i modules has its own controller, which can output any part of the internal memory. The part of the memory as well as the output frequency can be set for each channel separately.

To configure the individual channels, the parameters are registered in the **`struct ad_mda2_generator`** structure and are passed by **`ad_ioctl()`**.

The **struc ad_mda2_generator** structure allows the definition of the parameters for all output channels of a module and looks like as follows:

```c
struct ad_mda2_generator
{
  uint32_t cha;
  uint32_t chac;
  struct ad_mda2_generator_cha chav[16];
  uint32_t ram[2048];
};
```

The elements of the structure bear the following meaning:

cha

Output channel of the module: Defines for which module the output parameters are to be modified, e.g. for the first module on a DAQ card (**AD_CHA_TYPE_ANALOG_OUT|0x0001**) and for a possible second module (**AD_CHA_TYPE_ANALOG_OUT|0x0101**).

chac

Number of output controllers to be defined

chav

Output parameter structurs of the output controllers to be defined

ram

Memory with output values for the analog output: The analog output is linearily scaled. The value **0x00000000** of an analog output relates to the lowest output voltage, the value **0xffffffff** to the highest output voltage. With **ad_float_to_sample()** a voltage value (float) can be converted into an output value.

The **struc ad_mda2_generator_cha** structure allows the definition of the parameters for one output channel and looks like as follows:

```c
struct ad_mda2_generator_cha
{
  uint32_t cha;
  uint32_t range;
  uint32_t rate;
  uint32_t start_addr;
  uint32_t stop_addr;
};
```

The output controller **cha** periodically outputs storage data from the start address **start_addr** to the stop address **stop_addr** at the output channel considering the defined output rate **rate**. The respecting start and stop addresses of the output controller must not overlap.

The elements of the structure bear the following meaning:

cha

Controller number of the output channel: Each module features one controller per analog output. The controller number is starts with 0 (e.g. the controller numbers 0 to 3 are provided for the MDA16-4i providing 4 output channels). The controller number 0 relates to analog output 1, etc.

range

 © BMC Messsysteme GmbH

Measuring range number of the output: The measuring range number of the MDA16-2/4/8i with ±10.24V range is 0.

rate

Divisor for the output frequency: The output controller is operated with an output frequency resulting from the maximum output frequency of the module divided by **rate**. The maximum output frequency of the MDA16-2/4/8i is 100kHz. If **rate** is 100 the output resolution would be 1kHz or 1ms per output point.

start_addr

Start address of the module's storage range

stop_addr

Stop address of the module's storage range

The following example shows the basic procedure:

Prototype

```
int32_t
ad_ioctl (int32_t adh, int32_t ioc,
          void *par, int32_t size);
```

C

```
#include "libad.h"
#include "libad_mda2.h"
...

  struct ad_mda2_generator gen;
  unsigned j, N = 1000;
  float v;
  double PI = 3.141;
  int rc;
  uint32_t tmp;

  memset(&gen, 0, sizeof(gen));

  /* define the analog output modul
   /
  gen.cha = AD_CHA_TYPE_ANALOG_OUT|1;

  /* using 2 output controller
   */
  gen.chac = 2;

  /* fill two areas in the modul ram,
   * 1st area 0..499 with a full sinus,
   * 2nd area 500 to 999 with a ramp
   */
  for (j = 0; j < 500; j++)
    {
      v = (float) (10*sin (j * ((2.0*PI) / 500)));
      ad_float_to_sample(adh, AD_CHA_TYPE_ANALOG_OUT|1, 0, v, &tmp);
      gen.ram[j] = tmp;
    }

  for (j = 500; j < 1000; j++)
    {
      v = (float) (-1.0 + (j-500) * 2.0 / 500);
      ad_float_to_sample(adh, AD_CHA_TYPE_ANALOG_OUT|2, 0, v, &tmp);
      gen.ram[j] = tmp;
    }

  gen.chav[0].cha = 0;
  /* rate 10kHz (100kHz/10) with 500 points
   * => 50 ms duration */
  gen.chav[0].rate = 10;
  gen.chav[0].start_addr = 0;
  gen.chav[0].stop_addr = 499;

  gen.chav[1].cha = 1;
  /* rate 1kHz (100kHz/100) with 500 points
   * => 500 ms duration */
  gen.chav[1].rate = 100;
  gen.chav[1].start_addr = 500;
  gen.chav[1].stop_addr = 999;

  rc = ad_ioctl (adh, AD_MDA2_SET_GENERATOR, &gen, sizeof(gen));

  rc = ad_ioctl (adh, AD_MDA2_START_GENERATOR, &gen, sizeof(gen));
```


© BMC Messsysteme GmbH

6.4 USB-AD / USB-AD-OEM / USB-PIO / USB-PIO-OEM

Open the USB-AD or the USB-PIO/USB-PIO-OEM with the **LIBAD4** by passing the string "**usb-ad**" or "**usb-pio**" to **ad_open()**. To distinguish between several USB data acquisition systems, the device number is explicitly used (e.g. 1st device with "**usb-ad:0**", 2nd device with "**usb-ad:1**", etc., or 1st device with "**usb-pio:0**", 2nd device with "**usb-pio:1**", etc.). The device order results from the order of connecting.

As USB data acquisition systems can be plugged and unplugged during operation, it may happen that the device numbers are not assigned consecutively. For example, if the second of three connected USB-AD devices is removed, the remaining USB-AD devices are addressed with "**usb-ad:0**" and "**usb-ad:2**".

To avoid managing the order of connecting, a device is also accessible via its serial number. The device with the serial number 157 can be addressed with "**usb-ad:@157**" or "**usb-pio:@157**", for example.

6.4.1 Key Data / Channel Numbers USB-AD

DAQ system	Analog	Channel number	Measuring range	Output range	Digital	Direction
USB-AD	16 inputs 1 outputs	1..16 1	0 (±5.12V)	0 (±5.12V)	2 Ports (4 bit each)	1: input (bit 0..3) 2: output (bit 0..3)

The first analog input channel of a USB-AD starts with 1. The 16 analog inputs are defined by the following constants:

C

```
#define AI1    (AD_CHA_TYPE_ANALOG_IN|0x0001)
#define AI2    (AD_CHA_TYPE_ANALOG_IN|0x0002)
...
#define AI16   (AD_CHA_TYPE_ANALOG_IN|0x0010)
```

The analog output channel of a USB-AD is addressed by the following constant:

C

```
#define AO1    (AD_CHA_TYPE_ANALOG_OUT|0x0001)
```

For compatibility reasons, the measuring range 33 can be used for analog inputs and the output range 1 for the analog output.

The direction of the digital ports is hard-wired. The 4 lines of the first port (DIO1) are set to input, the 4 lines of the second port (DIO2) to output.

The USB-AD-OEM provides two ports with 8 lines each. The port direction is switchable (for all 8 lines of a port each) The first port (DIO1) iss et on input by default, the second port (DIO2) iss et on output by default.

The following constants result:

C

```
#define DIO1    (AD_CHA_TYPE_DIGITAL_IO|0x0001)
#define DIO2    (AD_CHA_TYPE_DIGITAL_IO|0x0002)
```

6.4.2 Key Data / Channel Numbers USB-PIO(-OEM)

DAQ system	Digital	Channel numberr
USB-PIO, USB-PIO-OEM	3 ports (8 bit each)	1..3 (bit 0..7)

The line direction is set separately for each port in groups of eight (see "**`ad_set_line_direction`**", p. 20).

C

```
#define DIO1    (AD_CHA_TYPE_DIGITAL_IO|0x0001)
#define DIO2    (AD_CHA_TYPE_DIGITAL_IO|0x0002)
#define DIO3    (AD_CHA_TYPE_DIGITAL_IO|0x0003)
```

6.5 USB-AD14f

Open the USB-AD14f with the **LIBAD4** by passing the string "**`usbad14f`**" or "**`usbad12f`**" to **`ad_open ()`**. To distinguish between several USB data acquisition systems, the device number is explicitly used (e.g. 1st USB-AD14f with "**`usbad14f:0`**", 2nd USB-AD14f with "**`usbad14f:1`**", etc.). The device order results from the order of connecting.

As USB data acquisition systems can be plugged and unplugged during operation, it may happen that the device numbers are not assigned consecutively. For example, if the second of three connected USB-AD14f devices is removed, the remaining USB-AD14f devices are addressed with "**`usbad14f:0`**" and "**`usbad14f:2`**".

To avoid managing the order of connecting, a device is also accessible via its serial number. The USB-AD14f with the serial number 157 can be addressed with "**`usbad14f:@157`**", for example.

6.5.1 Key Data / Channel Numbers USB-AD14f

DAQ system	Analog	Channel number	Measuring range	Output range	Digital	Direction
USB-AD14f	16 inputs 1 output	1..16 1	0 (±10.24V)	0 (±5.12V)	2 ports (8 bit each)	1: input (bit 0..7) 2: output (bit 0..7)

The first analog input channel of a USB-AD14f starts with 1. The 16 analog inputs are defined by the following constants:

© BMC Messsysteme GmbH

C

```
#define AI1   (AD_CHA_TYPE_ANALOG_IN|0x0001)
#define AI2   (AD_CHA_TYPE_ANALOG_IN|0x0002)
...
#define AI16  (AD_CHA_TYPE_ANALOG_IN|0x0010)
```

The analog output channel of a USB-AD14f is addressed by the following constant:

C

```
#define AO1   (AD_CHA_TYPE_ANALOG_OUT|0x0001)
```

The direction of the digital ports is hard-wired. The 8 (USB-AD14f) lines of the first port (DIO1) are set to input, the 8 (USB-AD14f) or lines of the second port (DIO2) to output. The following constants result:

C

```
#define DIO1   (AD_CHA_TYPE_DIGITAL_IO|0x0001)
#define DIO2   (AD_CHA_TYPE_DIGITAL_IO|0x0002)
```

The digital input line 1 can also be used as a counter input. The counter is addressed by the following channel constant:

C

```
#define CNT1   (AD_CHA_TYPE_COUNTER|0x0001)
```

6.6 USB-AD16f / AMS42/84-USB

Open the USB-AD16f or AMS42/84-USB with the **LIBAD4** by passing the string "**`usbbase`**" to **`ad_open ()`**. To distinguish between several USB data acquisition systems, the device number is explicitly used (e.g. 1st device with "**`usbbase:0`**", 2nd device with "**`usbbase:1`**", etc.). The device order results from the order of connecting.

As USB data acquisition systems can be plugged and unplugged during operation, it may happen that the device numbers are not assigned consecutively. For example, if the second of three connected USB-AD16f or AMS42/84-USB devices is removed, the remaining devices are addressed with "**`usbbase:0`**" and "**`usbbase:2`**".

To avoid managing the order of connecting, a device is also accessible via its serial number. The device with the serial number 157 can be addressed with "**`usbbase:@157`**", for example.

6.6.1 Key Data / Channel Numbers USB-AD16f / AMS42/84-USB

DAQ system	Analog	Channel number	Measuring range	Output range	Digital	Direction
USB-AD16f / AMS42/84-USB	16 inputs 2 output	1..16 1 .. 2	0 (±1.024V) 1 (±2.048V) 2 (±5.12V) 3 (±10.24V)	0 (±10.24V)	2 oorts (4 bit each)	1: input (bit 0..3) 2: output (bit 0..3)

The 16 analog inputs of a USB-AD16f or AMS42/84-USB are addressed via the channel numbers 1-16. The 2 analog outputs are reached via channel number 1 and 2.

The 16 analog inputs are defined by the following constants:

C

```c
#define AI1    (AD_CHA_TYPE_ANALOG_IN|0x0001)
#define AI2    (AD_CHA_TYPE_ANALOG_IN|0x0002)
...
#define AI16   (AD_CHA_TYPE_ANALOG_IN|0x0010)
```

The two analog output channels of a USB-AD16f or AMS42/84-USB are addressed by the following constants:

C

```c
#define AO1    (AD_CHA_TYPE_ANALOG_OUT|0x0001)
#define AO2    (AD_CHA_TYPE_ANALOG_OUT|0x0002)
```

The direction of the digital ports is hard-wired. The 4 lines of the first port (DIO1) are set to input, the 4 lines of the second port (DIO2) to output. The following constants result:

C

```c
#define DIO1    (AD_CHA_TYPE_DIGITAL_IO|0x0001)
#define DIO2    (AD_CHA_TYPE_DIGITAL_IO|0x0002)
```

Besides that, the USB-AD16f and AMS42/84-USB features a counter input, which is defined as follows:

C

```c
#define CNT1     (AD_CHA_TYPE_COUNTER|0x0001)
```

6.7 USB-OI16

Open the USB-OI16 with the **LIBAD4** by passing the string "`usb-oi16`" to `ad_open()`. To distinguish between several USB devices, the device number is explicitly used (e.g. 1st device with "`usb-oi16:0`", 2nd device with "`usb-oi16:1`", etc.). The device order results from the order of connecting.

© BMC Messsysteme GmbH

As USB data acquisition systems can be plugged and unplugged during operation, it may happen that the device numbers are not assigned consecutively. For example, if the second of three connected USB-OI16 devices is removed, the remaining USB-OI16 devices are addressed with "**`usb-oi16:0`**" and "**`usb-oi16:2`**".

To avoid managing the order of connecting, a device is also accessible via its serial number. The device with the serial number 157 can be addressed with "**`usb-oi16:@157`**", for example.

6.7.1 Key Data / Channel Numbers USB-OI16

DAQ system	Digital	Channel number
USB-OI16	2 ports (16 bit each)	1: input 2: output

6.7.2 Channel numbers USB-OI16

The USB-OI16 provides two 16-bit digital ports. The direction of the digital ports is hard-wired. The 16 lines of the first port (DIO1) are set to input, the 16 lines of the second port (DIO2) to output. The following constants result:

C

```c
#define DIO1  (AD_CHA_TYPE_DIGITAL_IO|0x0001)
#define DIO2  (AD_CHA_TYPE_DIGITAL_IO|0x0002)
```

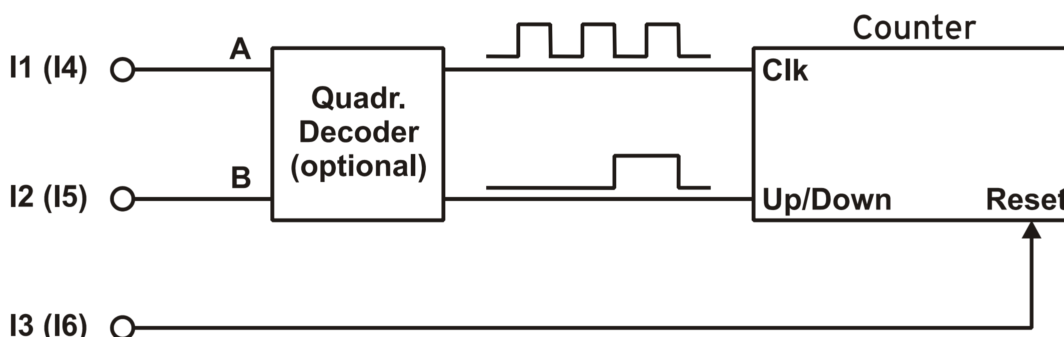
Besides that, the USB-OI16 features two 32-bit counter inputs. They can be used in different operating modes and must be configured via software before use (see "Configuration of the USB-OI16 Counters", p. 64). The inputs of the counter (Signal A, Signal B, Reset) are connected to the first digital input lines of the USB-OI16 digital port.

The following constants are defined for the 32-bit counter input:

C

```c
#define CNT1   (AD_CHA_TYPE_COUNTER|0x0001)
#define CNT2   (AD_CHA_TYPE_COUNTER|0x0002)
```

6.7.3 Configuration of the USB-OI16 Counters



For counter settings, the configuration parameters are entered in the **`struct ad_counter_mode`** structure and passed to **`ad_ioctl()`**.

The following example demonstrates the general procedure: It configures the first counter of the USB-OI16 in the "Counter" operating mode.

Prototype

```
int32_t
ad_ioctl (int32_t adh, int32_t ioc,
          void *par, int32_t size);
```

C

```
#include "libad.h"
...

struct ad_counter_mode par;
int32_t adh;
int32_t st;

...

  adh = ad_open ("pcibase");

  memset (&par, 0, sizeof(par));
  par.cha = AD_CHA_TYPE_COUNTER|1;
  par.mode = AD_CNT_COUNTER;
  st = ad_ioctl (adh, AD_SET_COUNTER_MODE,
                 &par, sizeof(par));
...
  ad_close (adh);
```

The elements of the structure bear the following meaning:

cha

Determines the counter channel to be configured.

mode

Sets the operating mode of the counter.

Operating mode	Description
AD_CNT_COUNTER	The counter channel is used as a simple counter. Input A of the counter is used only. Each positive edge at the input increases the counter.
AD_CNT_UPDOWN	The counter channel is used as an Up/Down counter, i.e. the counter is bidirectional. Input A of the counter is for the pulse input, input B for changing the direction. If input B of the counter is low, each positive edge at input A increases the counter. Otherwise, the positive edge reduces the counter.
AD_CNT_QUAD_DECODER	The counter decodes the two tracks of an incremental encoder. In this case, each edge of the two tracks is decoded.
AD_CNT_PULSE_TIME	Configures the counter for pulse time measurement. In this case, the counter input is connected with an internal clock (60MHz) and will be started and stopped at each edge of input A.

mux_a, mux_b, mux_rst

© BMC Messsysteme GmbH

Defines the pins of the two digital ports that are connected to the inputs of the counter. It is not possible to connect the counter inputs with different digital ports (i.e. inputs A, B and *Reset* must either all be connected with pins of port A or all with pins of port B).

`mux_a`, `mux_b` or `mux_rst`	Port/Line
0	PA/1
1	PA/2
2	PA/3
3	PA/4
4	PA/5
5	PA/6
6	PA/7
7	PA/8
8	PA/9
9	PA/10
10	PA/11
11	PA/12
12	PA/13
13	PA/14
14	PA/15
15	PA/16

`mux_a`, `mux_b` or `mux_rst`	Port/Line
16	PB/1
17	PB/2
18	PB/3
19	PB/4
20	PB/5
21	PB/6
22	PB/7
23	PB/8
24	PB/9
25	PB/10
26	PB/11
27	PB/12
28	PB/13
29	PB/14
30	PB/15
31	PB/16

`flags`

Determines the operation mode of the counter inputs. The operation modes can be combined with **`OR`**: e.g. **`AD_CNT_INV_RST|AD_CNT_ENABLE_RST`**.

Operating mode	Description
AD_CNT_INV_A	Counter input A reacts inversely.
AD_CNT_INV_B	Counter input B reacts inversely.
AD_CNT_INV_RST	Reset input reacts inversely.
AD_CNT_ENABLE_RST	Reset input is activated.

7 Index

A

ad_analog_in () 19
ad_analog_out () 19
ad_calc_run_size () 44
ad_close () 8
ad_digital_in () 20
ad_digital_out () 20
ad_discrete_in () 10
ad_discrete_in64 () 11
ad_discrete_inv () 12
ad_discrete_out () 13, 14
ad_discrete_outv () 15
ad_float_to_sample () 18, 19
ad_get_digital_line () 20
ad_get_drv_version () 21
ad_get_line_direction () 21
ad_get_next_run () 45
ad_get_next_run_f () 45
ad_get_next_run_f64 () 46
ad_get_product_info () 22
ad_get_range_count () 8
ad_get_range_info () 9
ad_get_sample_layout () 41
ad_get_samples_f () 43
ad_get_samples_f64 () 42, 43
ad_get_version () 21
ad_open () 4, 6
ad_poll_scan_state () 46
ad_sample_to_float () 16
ad_sample_to_float64 () 17
ad_set_digital_line () 20
ad_set_line_direction () 21
ad_start_mem_scan () 40
ad_start_scan () 41
ad_stop_scan () 46
AMS42-LAN16f 47
AMS42-LAN16fx 47
AMS42-USB 63
AMS84-LAN16f 47
AMS84-LAN16fx 47
AMS84-USB 63
Analog output
  Set 13, 14
  Set several 15

B

Buffer 5, 23, 26, 30, 44
buffer_start 41
bytes_per_run 26, 44

C

Case sensitivity 4, 6
cha 23, 50, 54, 58, 66
chac 58
Channel number 10, 11, 12, 13, 14, 15, 23, 24, 47
Channel type 47
chav 58
Continuous scan 26, 31
Conversion
  Measuring value into voltage value 16, 17
  Voltage value into measuring value 18
Copyright 2
Counter 51, 55, 67
  Configuration 49, 53, 66
Counter channel 50, 54, 66

D

Data acquisition system
  Close 4, 8
  Name 4
  Open 4, 6
  Open several different ones 4, 6
  Open several equal ones 7
Digital channel
  Get direction 21
  Set direction 21
Direction 21
Driver version 21

E

Error number 4, 6, 46

F

Firmware version 22
flags 27, 51, 55, 67
FreeBSD 1

G

GetLastError 4, 6

H

Header file 4

I

Incremental encoder 51, 55, 67
Input
  Direction 21
Input line 21
Inputs
  Order 40, 41

© BMC Messsysteme GmbH

L

LAN-AD16f 47
LAN-AD16fx 47
  Channel number 47
  Counter 48, 49
  Digital ports 48
Linux 1

M

Mac OS X 1
MADDA16 56
MADDA16n 56
Maximum 24
MDA12 56
MDA12-4 56
MDA16 56
MDA16-2i 56
MDA16-4i 56
MDA16-8i 56
Mean value 24
Measuring range 10, 11, 12, 24, 47
  Information 9
  Middle 10, 11
  Number 8
Measuring value 13, 14, 16, 17, 18, 45
  Read out 36
Memory management of the measuring values
  Internal 36, 46
Memory-only scan 29, 40, 45, 46
Messwert 19
Minimum 24
mode 50, 54, 66
mux_a 51, 55, 67
mux_b 51, 55, 67
mux_rst 51, 55, 67

N

Name 4
Network byte order 45
Number of measuring values 23, 24, 27, 31

O

OR operator (|) 47
Output
  Direction 21
  Set 13, 14
  Set several 15
Output line 21
Output range 13, 14, 15, 47
Overrun of the samples 23, 31, 46

P

PCI cards
  Serial number 52
PCI-BASE1000 52
PCI-BASE300 52
PCI-BASEII 52
  Counter 53
  Digital ports 52
PCIe cards
  Serial number 52
PCIe-BASE 52
  Counter 53
  Digital ports 52
PCI-PIO 52
  Counter 53
  Digital ports 52
Pointer 30, 40
posthist 26, 27
posthist_samples 42
Posthistory 26, 36
  Number of measuring values 42
prehist 26
prehist_samples 42
Prehistory 26, 36
  First measuring value 42
  Number of measuring values 42
Product information 22
Product name 22
Pulse time measurement 51, 67

Q

Quadrature decoder 51, 55, 67

R

ram 58
range 24, 59
Range limit 10, 45
rate 59
ratio 24
Reading out measuring values 30
Result 46
RMS 24
Root mean square value 24
RUN 27, 31
runs_pending 27

S

Sample rate 35
sample_rate 26, 44
samples_per_run 24, 27, 44
Sampling period 23, 44
Sampling pulse 35
Sampling rate 25, 26, 31
Scan 5, 23
  Continuous 23, 31
  First measuring value 41
  Memory-only 23
  Parameters 23
  Start 29
  State 27, 36, 46
  Stop 31
  With trigger 36
Serial number 22, 52, 61, 62, 63, 65

© BMC Messsysteme GmbH

Single value
- Read ... 10, 11
- Read several ... 12

Spannungswert ... 19
start ... 42
Start scan ... 29
start_addr ... 59
Stop scan ... 31, 46
stop_addr ... 59
Storage
- Interval ... 24
- Ratio ... 35
- Type ... 24

store ... 24
struct ad_scan_cha_desc ... 23
struct ad_scan_desc ... 26
struct ad_scan_state ... 27

T

ticks_per_run ... 26, 44
trg_mode ... 24
trg_par ... 24
Trigger ... 24, 25, 26, 27, 36
- Condition ... 25, 36
- Negative Edge ... 26
- Parameters ... 24
- Positive Edge ... 26
- Settings ... 23
- Window ... 26

U

Umrechnung
- Spannungswert Messwert ... 19

Up/Down counter ... 51, 55, 67
USB-
- Order ... 61

USB-AD ... 61
- Channel number ... 61
- Digital ports ... 61
- Order ... 61
- Serial number ... 61

USB-AD12f ... 62
- Channel number ... 62
- Digital ports ... 63
- Order ... 62
- Serial number ... 62

USB-AD14f ... 62
- Channel number ... 62
- Digital ports ... 63
- Order ... 62
- Serial number ... 62

USB-AD16f ... 63
- Channel number ... 64
- Counter ... 64
- Digital ports ... 64
- Order ... 63
- Serial number ... 63

USB-AD-OEM
- Digitalports ... 61

USB-OI16 ... 64
- Channel number ... 65
- Counter ... 66
- Digital ports ... 65
- Order ... 64
- Serial number ... 65

USB-PIO ... 61
- Channel number ... 62
- Digital ports ... 62
- Direction ... 62
- Serial number ... 61

USB-PIO-OEM ... 61
- Channel number ... 62
- Digital ports ... 62
- Direction ... 62
- Serial number ... 61

V

Version
- Driver ... 21
- LIBAD4.DLL ... 21

Voltage value ... 16, 17, 18, 30

W

Window trigger ... 26
Windows ... 1

Z

zero ... 24
Zero level ... 24

© BMC Messsysteme GmbH