

LIBAD4

Library for Programming Interface LIBAD4

Programming Guide

Version 4.5

▶ www.bmcm.de

bavarian measurement company munich



Contents

1 Overview	7
1.1 Introduction	7
1.2 BMC Messsysteme GmbH	9
1.3 Copyrights	10
2 Installation	11
2.1 Installation under Windows®	11
2.2 Installation under Mac OS X	11
2.3 Installation under FreeBSD	14
2.4 Installation under Linux	16
2.5 Sharing the library	18
3 Basics	19
3.1 General	19
4 Single-value acquisition	21
4.1 Function description (single values)	21
4.1.1 ad_open	21
4.1.2 ad_close	24
4.1.3 ad_get_range_count	24
4.1.4 ad_get_range_info	25
4.1.5 ad_discrete_in	26
4.1.6 ad_discrete_in64	27
4.1.7 ad_discrete_inv	28
4.1.8 ad_discrete_out	30
4.1.9 ad_discrete_out64	31
4.1.10 ad_discrete_outv	32
4.1.11 ad_sample_to_float	33
4.1.12 ad_sample_to_float64	34
4.1.13 ad_float_to_sample	35

4.1.14	ad_float_to_sample64	36
4.1.15	ad_analog_in	37
4.1.16	ad_analog_out	37
4.1.17	ad_digital_in	38
4.1.18	ad_digital_out	38
4.1.19	ad_set_digital_line	38
4.1.20	ad_get_digital_line	39
4.1.21	ad_get_line_direction	39
4.1.22	ad_set_line_direction	40
4.1.23	ad_get_version	40
4.1.24	ad_get_drv_version	41

5 Scan process 42

5.1	Notes	42
5.2	Scan parameters	42
5.2.1	struct ad_scan_cha_desc	43
5.2.1.1	Storage types	44
5.2.1.2	Trigger types	45
5.2.2	struct ad_scan_desc	46
5.2.3	struct ad_scan_state	48
5.3	Memory-only Scan	49
5.3.1	Starting a scan	50
5.3.2	Reading out measuring values	51
5.3.3	Stopping a scan	53
5.4	Continuous scan	54
5.4.1	Composition of a RUN	54
5.4.2	One sample per RUN	56
5.4.3	Signals with different storage ratio	58
5.5	Scan with triggering	60
5.6	Function description (Scan)	61
5.6.1	ad_start_mem_scan	61
5.6.2	ad_start_scan	62
5.6.3	ad_get_sample_layout	63
5.6.4	ad_get_samples	64
5.6.5	ad_get_samples_f	65
5.6.6	ad_get_samples_f64	66
5.6.7	ad_calc_run_size	67

5.6.8	ad_get_next_run	68
5.6.9	ad_get_next_run_f	69
5.6.10	ad_get_next_run_f64	70
5.6.11	ad_poll_scan_state	70
5.6.12	ad_stop_scan	71
6	Data acquisition systems	72
6.1	Notes	72
6.2	iM-AD25a / iM-AD25 / iM3250T / iM3250	73
6.2.1	Channel numbers iM-AD25a / iM-AD25	74
6.2.2	Channel numbers iM3250T	74
6.2.3	Channel numbers iM3250	75
6.3	LAN-AD16f	76
6.3.1	Channel numbers LAN-AD16f	76
6.3.2	Configuration of the counter	78
6.4	PCIe-BASE / PCI-BASEII/300/1000/PIO	81
6.4.1	Digital ports and counters	81
6.4.1.1	PCIe-BASE	81
6.4.1.2	PCI-BASEII / PCI-PIO	82
6.4.1.3	PCI-BASE300 / PCI-BASE1000	82
6.4.1.4	Configuration of the counters	83
6.4.2	Plug-on modules	87
6.4.2.1	MAD12/12a/12b/12f/16/16a/16b/16f	87
6.4.2.2	MADDA16/16n	89
6.4.2.3	MDA12/12-4/16/16-2i/16-4i/16-8i	90
6.4.2.4	Function generator of the MDA16i-2i/-4i/-8i	91
6.5	meM-AD /-ADDA /-ADf / -ADfo	96
6.5.1	Key data and channel numbers meM devices	96
6.6	meM-PIO / meM-PIO-OEM	98
6.6.1	Key data and channel numbers meM-PIO(-OEM)	98
6.7	USB-AD / USB-PIO / USB-PIO-OEM	99
6.7.1	Key data and channel numbers USB-AD	100
6.7.2	Key data and channel numbers USB-PIO(-OEM)	102
6.8	USB-AD12f	102
6.8.1	Key data and channel numbers USB-AD12f	103
6.9	USB-AD16f	104
6.9.1	Key data and channel numbers USB-AD16f	104

1 Overview

1.1 Introduction

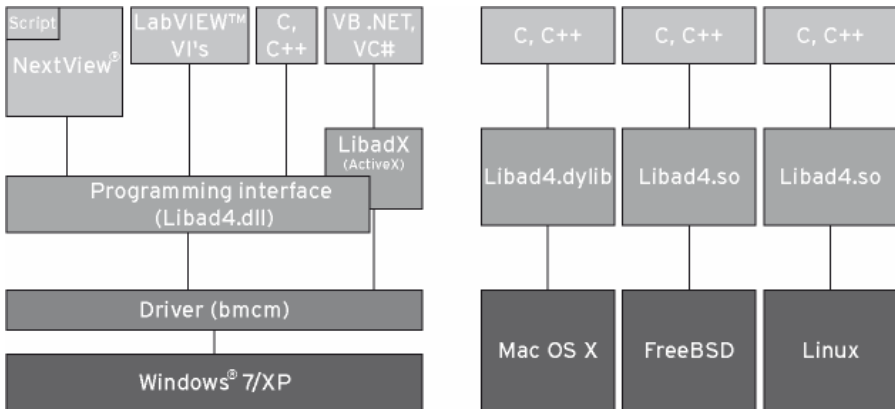


Figure 1

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 losing any measuring values.

The **LIBAD4** is provided for Windows[®] 7/XP 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 (<http://www.bmcm.de/us>) for detailed information and latest news.



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 © 2011

Updated: 10/11/2011

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 Installation under Mac OS X



Under Mac OS X, the **LIBAD4** is implemented as "dynamic link library and is delivered as a disk image.

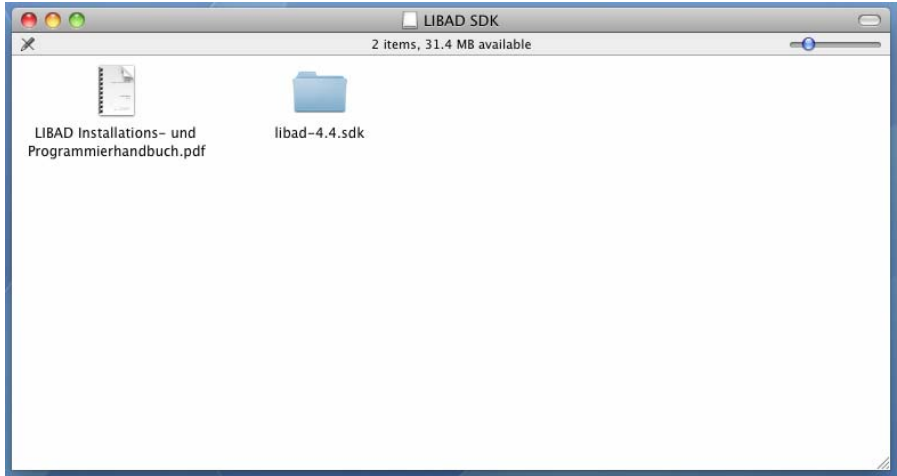


Figure 2

In addition to the installation and programming guide, the header files, the library itself, and the example programs are also provided in the disk image.

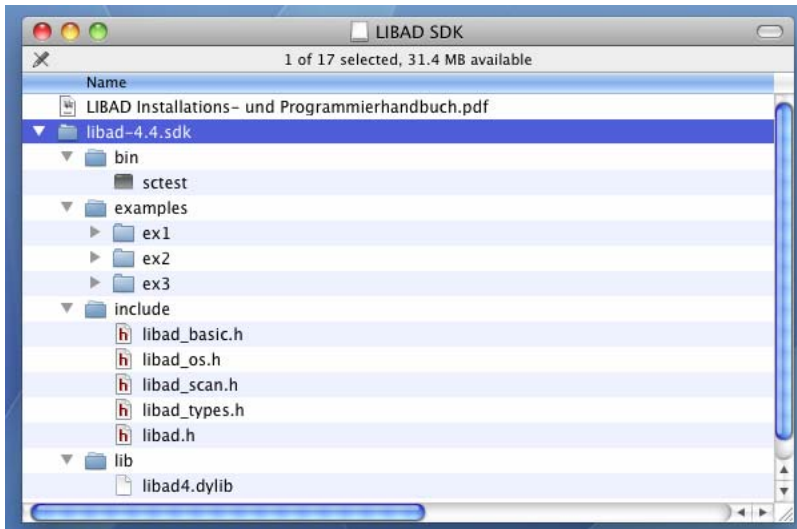


Figure 3

The `libad4.dll` should be copied into a directory in which the dynamic linker expects shared libraries in order for the programs to access the library. Please see the manpage of `dyld` for details.

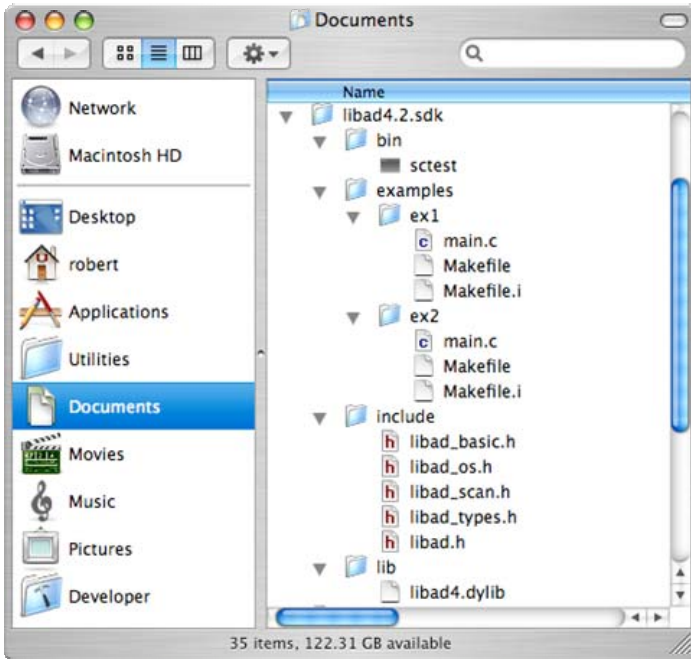


Figure 4

The following command copies the `LIBAD4` to `/usr/local/lib`:

```
root# cp lib/libad.dylib /usr/local/lib
root#
```

2.3 Installation under FreeBSD



Under FreeBSD, the **LIBAD4** is provided as a packed TAR file, which is unpacked with the following command (please use the version number of the used LIBAD).

```
bash# tar xjf libad-freebsd-4.4.412.tar.bz2
bash#
```

The following files are on hard disc after unpacking:

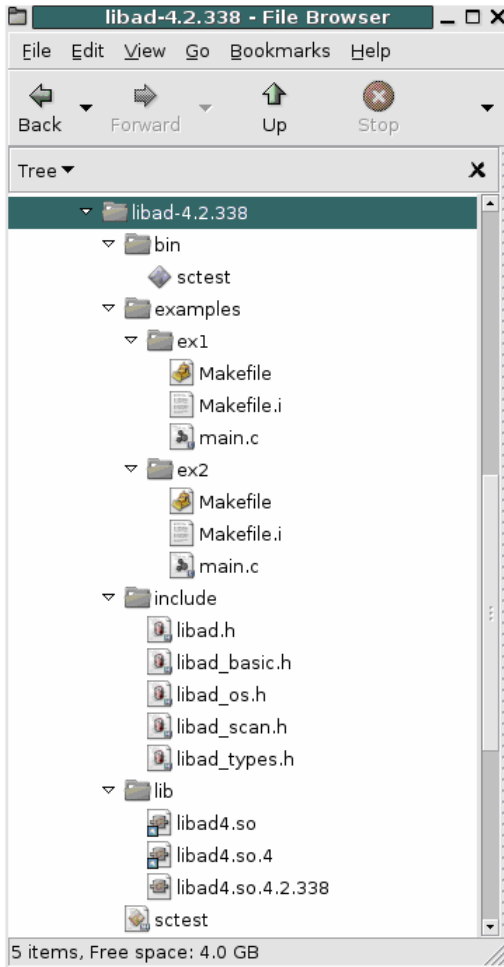


Figure5

Under FreeBSD, the **LIBAD4** is implemented as "shared library". It should be copied into a directory in which the dynamic linker expects shared libraries. Please see the manpage of **ldconfig** or **ld-elf.so.1** for detailed information.

If your system uses shared libraries in `/usr/local/lib`, please copy the **libad4.so.4.4.412** to `/usr/local/lib`. Then create two symbolic links `/usr/local/lib/libad4.so.4` and `/usr/local/lib/libad4.so`

pointing to `/usr/local/lib/libad4.so.4.4.412` (if the version number of your **LIBAD4** differs, it must be adapted accordingly).

The following commands perform the necessary actions:

```
bash# cp lib/libad4.so.4.4.412 /usr/local/lib/libad4.so.4.4.412
bash# ln -sf libad4.so.4.4.412 /usr/local/lib/libad4.so.4
bash# ln -sf libad4.so.4.4.412 /usr/local/lib/libad4.so
bash# /sbin/ldconfig
bash#
```

2.4 Installation under Linux



Under Linux, the **LIBAD4** is provided as a packed TAR file, which is unpacked with the following command (please use the version number of the LIBAD used).

```
bash# tar xjf libad-linux-4.4.412.tar.bz2
bash#
```

The following files are on hard disc after unpacking:

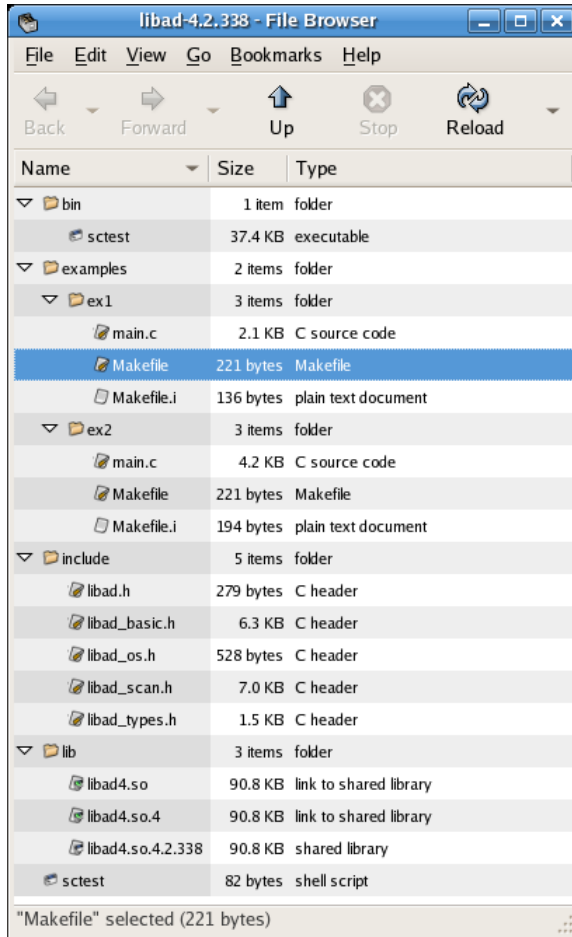


Figure6

Under Linux, the **LIBAD4** is implemented as "shared library". The library is translated for 32-bit systems. The library should be copied into a directory in which **ldconfig** expects shared libraries in order for the programs to access the library. Please see the manpage of **ldconfig** or the file `/etc/ld.so.conf` for details.

If your system uses shared libraries in `/usr/local/lib`, please copy the **libad4.so.4.4.412** to `/usr/local/lib`. Then create two symbolic links

`/usr/local/lib/libad4.so.4` and `/usr/local/lib/libad4.so` pointing to `/usr/local/lib/libad4.so.4.4.412` (if the version number of your **LIBAD4** differs, it must be adapted accordingly). The following commands perform the necessary actions:

```
bash# cp lib/libad4.so.4.4.412 /usr/local/lib/libad4.so.4.4.412
bash# ln -sf libad4.so.4.4.412 /usr/local/lib/libad4.so.4
bash# ln -sf libad4.so.4.4.412 /usr/local/lib/libad4.so
bash# /sbin/ldconfig
bash#
```

2.5 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 expressly permitted (if the version number of your **LIBAD4** differs, it must be adapted accordingly).

```
libad4.dll
libad4.dylib
libad4.so.4.4.412
```

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!**

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:



Prototype	<code>int32_t ad_open (const char *name);</code>
------------------	--

C	<pre>#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);</pre>
----------	---

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. 21) 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 72 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. 26) or output values can be set (see "`ad_discrete_out`", p. 30). Please see the chapter about "Single-value acquisition" on page 21 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 42 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.

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



Prototype	<pre>int32_t ad_open (const char *name);</pre>
------------------	--

C	<pre>#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);</pre>
----------	--

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 PCI-E-BASE / PCI-BASEII/300/1000/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:

- iM-AD25a / iM-AD25 / iM3250T / iM3250 / LAN-AD16f
- PCI-E-BASE / PCI-BASEII/300/1000/PIO
- meM-AD /-ADDA /-ADf / -ADfo / meM-PIO / meM-PIO-OEM
- USB-AD / USB-PIO / USB-PIO-OEM / USB-AD12f / USB-AD16f

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.



```

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	<pre>int32_t ad_close (int32_t adh);</pre>
------------------	--

C	<pre>#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);</pre>
----------	---

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	<pre>int32_t ad_get_range_count (int32_t adh, int32_t cha, int32_t cnt);</pre>
------------------	--

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

4.1.4 ad_get_range_info



Prototype	<pre> 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); </pre>
------------------	--

C	<pre> #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); </pre>
----------	--

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	<pre>int32_t ad_discrete_in (int32_t adh, int32_t cha, int32_t range, uint32_t *data);</pre>
------------------	--

C	<pre>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);</pre>
----------	--

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. 72). 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 **0x10000000** 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. 72).

4.1.6 ad_discrete_in64



Prototype	<pre>int32_t ad_discrete_in64 (int32_t adh, int32_t cha, uint64_t range, uint64_t *data)</pre>
------------------	---

C	<pre>int32_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);</pre>
----------	---

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 **0x1000000000000000** (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. 72).

4.1.7 `ad_discrete_inv`



Prototype	<pre>int32_t ad_discrete_inv (int32_t adh, int32_t chac, int32_t chav[], uint64_t rangev[], uint64_t datav[]);</pre>
------------------	--

```

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	<pre>int32_t ad_discrete_out (int32_t adh, int32_t cha, int32_t range, uint32_t data);</pre>
------------------	---

C	<pre>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);</pre>
----------	--

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. 72).

4.1.9 ad_discrete_out64



Prototype	<pre>int32_t ad_discrete_out64 (int32_t adh, int32_t cha, uint64_t range, uint64_t data);</pre>
------------------	---

C	<pre>int32_t adh; int32_t st; uint64_t data; ... adh = ad_open ("pci300"); 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);</pre>
----------	--

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 `0x1000000000000000` 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. 72).

4.1.10 ad_discrete_outv



Prototype	<pre>int32_t ad_discrete_outv (int32_t adh, int32_t chac, int32_t chav[], uint64_t rangev[], uint64_t datav[]);</pre>
------------------	---

C	<pre>#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);</pre>
----------	--

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.

4.1.11 `ad_sample_to_float`



Prototype	<pre>int32_t ad_sample_to_float (int32_t adh, int32_t cha, int32_t range, uint32_t data float *f);</pre>
------------------	--

C	<pre>int32_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);</pre>
----------	---

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. 72).

4.1.12 `ad_sample_to_float64`



Prototype	<pre>int32_t ad_sample_to_float64 (int32_t adh, int32_t cha, uint64_t range, uint64_t data double *dbl);</pre>
------------------	--

C	<pre>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);</pre>
----------	---

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. 72).

4.1.13 ad_float_to_sample



Prototype	<pre>int32_t ad_float_to_sample (int32_t adh, int32_t cha, int32_t range, float f, uint32_t *data);</pre>
------------------	---

C	<pre>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);</pre>
----------	---

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. 72).

4.1.14 ad_float_to_sample64



Prototype	<pre>int32_t ad_float_to_sample64 (int32_t adh, int32_t cha, uint64_t range, double dbl, uint64_t *data);</pre>
------------------	---

C	<pre>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);</pre>
----------	--

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. 72).

4.1.15 `ad_analog_in`



Prototype	<pre>int32_t ad_analog_in (int32_t adh, int32_t cha, int32_t range, float *volt);</pre>
------------------	---

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. 72).

4.1.16 `ad_analog_out`



Prototype	<pre>int32_t ad_analog_out (int32_t adh, int32_t cha, int32_t range, float volt);</pre>
------------------	--

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. 72).

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.

4.1.20 ad_get_digital_line



Prototype	<pre>int32_t ad_get_digital_line (int32_t adh, int32_t cha, int32_t line, uint32_t *flag);</pre>
------------------	--

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	<pre>int32_t ad_get_line_direction (int32_t adh, int32_t cha, uint32_t *mask);</pre>
------------------	--

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-AD12f).

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()**.

4.1.24 ad_get_drv_version



Prototype	<pre>int32_t ad_get_drv_version (int32_t adh, uint32_t *vers);</pre>
------------------	--

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

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 independently run a scan (e.g. iM-AD25a / iM-AD25 / iM3250T / iM3250 / LAN-AD16f, PCIe-BASE / PCI-BASEII/300/1000 with MAD/MADDA modules, USB-AD16f, USB-AD12f, meM-ADf, meM-ADfo). 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
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. 72).
- **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. 72).
- **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. 44).
- **ratio**
Defines the storage interval (see "Storage types", p. 44).
- **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 (in msec)	0	2	4	6	8	10	12	14	16	18	20	22	24	...
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	a ₁₃	...
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	a ₁₃	...

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      #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 (in msec)	0	2	4	6	8	10	12	14	16	18	20	22	24	...
Sample	a₁	a₂	a₃	a₄	a₅	a₆	a₇	a₈	a₉	a₁₀	a₁₁	a₁₂	a₁₃	...
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 (in msec)	0	2	4	6	8	10	12	14	16	18	20	22	24	...
Sample	a₁	a₂	a₃	a₄	a₅	a₆	a₇	a₈	a₉	a₁₀	a₁₁	a₁₂	a₁₃	...
Stored value					a₅ $\frac{1}{5}\sum a_i$					a₁₀ $\frac{1}{5}\sum a_i$...

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:

```
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. iM-AD25a / iM-AD25 / iM3250T / iM3250 / LAN-AD16f, USB-AD16f, USB-AD12f, PCIe-BASE / PCI-BASEII/300/1000 with MAD/MADDA modules, meM-ADf, meM-ADfo).
- 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()`.

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()`.

5.3 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 Starting a scan

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.



```
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      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
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	a₁	a₂	a₃	...	a₄₉₉	a₅₀₀	b₁	b₂	b₃	...	b₄₉₉	b₅₀₀

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.

The following example code stops the scan:



```
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 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 ms)	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 ms)	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

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



```

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

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



```

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;

          ...

```

The example above generates 500 RUNs with the following content:

Array index	0	1
Time (in msec)	0	0
Measuring value	a_1	b_1

RUN #0

Array index	0	1
Time (in msec)	10	10
Measuring value	a_2	b_2

RUN #1

Array index	0	1
Time (in msec)	4980	4980
Measuring value	a₄₉₉	b₄₉₉

RUN #498

Array index	0	1
Time (in msec)	4990	4990
Measuring value	a₅₀₀	b₅₀₀

RUN #499

5.4.3 Signals with different storage ratio

The two previous examples (see "One sample per RUN", p. 56) 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 be 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 ms)	0	2	4	6	8	10	12	14	16	18	20	22	24	...
Sample Input a	a₁	a₂	a₃	a₄	a₅	a₆	a₇	a₈	a₉	a₁₀	a₁₁	a₁₂	a₁₃	...
Sample Input b	b₁	b₂	b₃	b₄	b₅	b₆	b₇	b₈	b₉	b₁₀	b₁₁	b₁₂	b₁₃	...
Stored value Input a	a₁	a₂	a₃	a₄	a₅	a₆	a₇	a₈	a₉	a₁₀	a₁₁	a₁₂	a₁₃	...
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 msec)	0	2	4	6	8
Input a	a₁	a₂	a₃	a₄	a₅
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):

Array index	0	1	2	...	248	249	250	251	252	...	398	399
Time (in msec)	0	2	4	...	496	498	8	18	28	...	488	498
Values	a₁	a₂	a₃	...	a₂₄₉	a₂₅₀	$\frac{1}{5} \sum b_i$	$\frac{1}{5} \sum b_i$	$\frac{1}{5} \sum b_i$...	$\frac{1}{5} \sum b_i$	$\frac{1}{5} \sum b_i$

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.

5.6 Function description (Scan)

5.6.1 ad_start_mem_scan



Prototype	<pre>int32_t ad_start_mem_scan (int32_t adh, struct ad_scan_desc *scan_desc, uint32_t chac, struct ad_scan_cha_desc *chav);</pre>
------------------	---

C	<pre>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 */</pre>
----------	--

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. 67).

5.6.2 `ad_start_scan`



Prototype	<pre>int32_t ad_start_scan (int32_t adh, struct ad_scan_desc *scan_desc, uint32_t chac, struct ad_scan_cha_desc *chav);</pre>
------------------	--

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. 54).



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. 67).

5.6.3 `ad_get_sample_layout`



Prototype	<pre>int32_t ad_get_sample_layout (int32_t adh, int32_t index, struct ad_sample_layout *layout);</pre>
------------------	--

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	<pre>struct ad_sample_layout { uint64_t buffer_start; uint64_t start; uint64_t prehist_samples; uint64_t posthist_samples; };</pre>
----------	---

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	<pre>int32_t ad_get_samples (int32_t adh, int32_t index, int32_t type, uint64_t offset, uint32_t *n, void *buf);</pre>
------------------	--

Returns the measuring values for the scan channel **index** directly provided by the DAQ system if the internal memory management of the measuring values has been activated. The scan channel numbering starts with **index** = 0.

The DAQ system returns 16-bit or 32-bit measuring values depending on the memory depth. Starting from the position **offset**, **n** measuring values 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.

5.6.5 ad_get_samples_f



Prototype	<pre>int32_t ad_get_samples_f (int32_t adh, int32_t index, int32_t type, uint64_t offset, uint32_t *n, float *buf);</pre>
------------------	---

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. 19-bit counter of the LAN-AD16f or 32-bit counter of the PCIe-BASE / PCI-BASEII/300/1000/PIO).

5.6.6 ad_get_samples_f64



Prototype	<pre>int32_t ad_get_samples_f64 (int32_t adh, int32_t index, int32_t type, uint64_t offset, uint32_t *n, double *buf);</pre>
------------------	--

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	<pre>int32_t ad_calc_run_size (int32_t adh, struct ad_scan_desc *scan_desc, uint32_t chac, struct ad_scan_cha_desc *chav);</pre>
------------------	--

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
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);
```

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	<pre>int32_t ad_get_next_run_f (int32_t adh, struct ad_scan_state *state, uint32_t *run, float *p);</pre>
------------------	---

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. 19-bit counter of the LAN-AD16f or 32-bit counter of the PCIe-BASE / PCI-BASEII/300/1000/PIO).

5.6.10 `ad_get_next_run_f64`



Prototype	<pre>int32_t ad_get_next_run_f64 (int32_t adh, struct ad_scan_state *state, uint32_t *run, double *p);</pre>
------------------	--

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).

5.6.11 `ad_poll_scan_state`



Prototype	<pre>int32_t ad_poll_scan_state (int32_t adh, struct ad_scan_state *state);</pre>
------------------	--

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 been activated (`AD_SF_SAMPLES` bit of the `flags` element in the `struct ad_scan_desc` structure is set) the routine `ad_poll_scan_state()` must continuously be called.

5.6.12 `ad_stop_scan`



Prototype	<pre>int32_t ad_stop_scan (int32_t adh, int32_t *scan_result);</pre>
------------------	--

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 `struct ad_scan_desc()` was 0), it must be finished with `ad_stop_scan()`.

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-AD12f, 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 iM-AD25a / iM-AD25 / iM3250T / iM3250



To open the iM-AD25a, iM-AD25, iM3250T or iM3250 with the **LIBAD4**, the string "**im:<ip-addr>**" must be passed to **ad_open()**. Here **<ip-addr>** must be replaced by the relating IP address. The string "**im:192.168.1.1**", for example, opens the iM device with the IP address 192.168.1.1. When opening the driver, no difference is made between different iM device types.

DAQ syst.	Analog	Channel number	Meas. range	Range	Digital
iM-AD25a	16 inputs	1..16	$\pm 10.24V$ $\pm 5.12V$	1 0	1 output (bit 0..3)
iM-AD25	16 inputs	1..16	$\pm 5.12V$	0	1 output (bit 0..3)
iM3250T	32 inputs	17..48	$\pm 5.12V$	0	-
iM3250	32 inputs	AIn 1..16: 1..16 (with 1 BPL) 17..32 (with 2 BPL) AIn 17..32: 33..48	$\pm 5.00V$	0	-



Please note that MAL measuring amplifiers installed in the iM3250T might change the measuring range of the corresponding channels.

6.2.1 Channel numbers iM-AD25a / iM-AD25

The first analog input channel of an iM-AD25a / iM-AD25 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)
```

In addition, the iM-AD25a / iM-AD25 features a hard-wired digital output with four lines.

```
C      #define DOUT   (AD_CHA_TYPE_DIGITAL_IO|0x0001)
```

6.2.2 Channel numbers iM3250T

The first analog input channel of an iM3250T starts with 17. The 32 analog inputs are defined by the following constants:

```
C      #define AI1    (AD_CHA_TYPE_ANALOG_IN|0x0011)
      #define AI2    (AD_CHA_TYPE_ANALOG_IN|0x0012)
      ...
      #define AI32   (AD_CHA_TYPE_ANALOG_IN|0x0030)
```

6.2.3 Channel numbers iM3250

The channel numbers of the iM3250 depend on the configuration level of the hardware. If only one BPL is installed in the device, the first 16 channels are numbered from 1 to 16. If both BPLs are installed, the first 16 channels start from 17 to 32. The second 16 inputs can be addressed via channel numbers 33 to 48.

```

C      #ifndef BPL1      /* 1 bpl installed /

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

        #else          /* 2 bpl's installed /

        #define AI1      (AD_CHA_TYPE_ANALOG_IN|0x0011)
        #define AI2      (AD_CHA_TYPE_ANALOG_IN|0x0012)
        ...
        #define AI16     (AD_CHA_TYPE_ANALOG_IN|0x0020)

        #endif /* BPL1 */

        #define AI17     (AD_CHA_TYPE_ANALOG_IN|0x0021)
        #define AI18     (AD_CHA_TYPE_ANALOG_IN|0x0022)
        ...
        #define AI32     (AD_CHA_TYPE_ANALOG_IN|0x0030)

```

6.3 LAN-AD16f



Open the LAN-AD16f 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-AD16f. 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.

DAQ system	Analog	Channel number	Measuring range	Output range	Digital	Direction
LAN-AD16f	16 inputs 2 outputs	1..16 1 .. 2	0 ($\pm 1.024V$) 1 ($\pm 2.048V$) 2 ($\pm 5.120V$) 3 ($\pm 10.240V$)	0 ($\pm 10.24V$)	2 ports (16 bit each)	1: input (bit 0..15) 2: output (bit 0..15)

6.3.1 Channel numbers LAN-AD16f

The 16 analog inputs of a LAN-AD16f 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-AD16f 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-AD16f 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      #define DIO1   (AD_CHA_TYPE_DIGITAL_IO|0x0001)
          #define DIO2   (AD_CHA_TYPE_DIGITAL_IO|0x0002)

```

Besides that, the LAN-AD16f features a counter input. It can be used in different operating modes and must be configured via software before use (see "Configuration of the counter", p. 78). The inputs of the counter (Signal A, Signal B, Reset) are connected to the first three digital input lines of the LAN-AD16f digital port. These settings have to be configured also before using the counter (see "Configuration of the counters", p. 83)

The following constant is defined for the 19-bit counter input:

```

C      #define CNT1    (AD_CHA_TYPE_COUNTER|0x0001)

```

6.3.2 Configuration of the counter

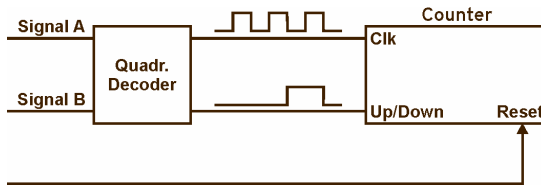


Figure7

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

The next example demonstrates the general procedure: It configures the counter of the LAN-AD16f with the serial number 157 in the "Counter" operating mode.

Prototype	<pre>int32_t ad_ioctl (int32_t adh, int32_t ioc, void *par, int32_t size);</pre>
------------------	--

C	<pre>#include "libad.h" ... struct ad_counter_mode par; int32_t adh; int32_t st; ... adh = ad_open ("lanbase:@157"); 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);</pre>
----------	--

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. If using the LAN-AD16f, this value always has to be "1".
- **mode**
Sets the operation mode of the counter.

Operationng 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.

➤ **mux_a, mux_b, mux_rst**

Defines the pins of the two digital ports that are connected to the inputs of the counter. This element is ignored with the LAN-AD16f, as the connection of the counter inputs cannot be chosen.

➤ **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**.

Operationng 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.

6.4 PCIe-BASE / PCI-BASEII/300/1000/PIO



To open the PCIe-BASE, PCI-BASEII, PCI-BASE300, PCI-BASE1000 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.

6.4.1 Digital ports and counters

The PCIe-BASE / PCI-BASEII/300/1000/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. 83).

6.4.1.1 PCIe-BASE

The digital ports of the PCIe-BASE are bidirectional (see "**ad_set_line_direction**", p. 40). After boot-up, the first port is set to input, the second to output. The following numbering is used:

C	<pre>#define DIO1 (AD_CHA_TYPE_DIGITAL_IO 0x0001) #define DIO2 (AD_CHA_TYPE_DIGITAL_IO 0x0002)</pre>
----------	--

Besides that, the PCIe-BASE provides a 32-bit counter input:

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

6.4.1.2 PCI-BASEII / PCI-PIO

The digital ports of the PCI-BASEII / PCI-PIO are bidirectional (see "ad_set_line_direction", p. 40). After boot-up, the first port is set to input, the second to output. The following numbering is used:

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

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

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

6.4.1.3 PCI-BASE300 / PCI-BASE1000

The digital ports of the PCI-BASE300 / PCI-BASE1000 are hard-wired: the first port is set to input, the second port to output. The following numbering is used:

```
C      #define DIN     (AD_CHA_TYPE_DIGITAL_IO|0x0001)
      #define DOUT    (AD_CHA_TYPE_DIGITAL_IO|0x0002)
```

6.4.1.4 Configuration of the counters

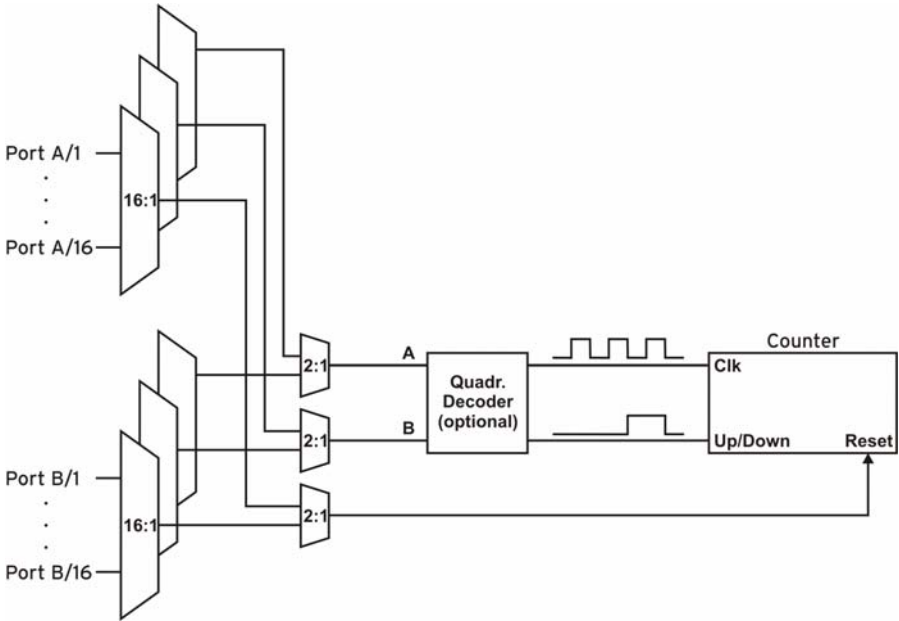


Figure8

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 PCI(e)-BASE card in the "Counter" operating mode and connects counter input A with the second input pin of the first digital port.

Prototype	<pre>int32_t ad_ioctl (int32_t adh, int32_t ioc, void *par, int32_t size);</pre>
------------------	--

C	<pre>#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);</pre>
----------	--

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

C	<pre>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; ... };</pre>
----------	--

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.

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

Operationng 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.

6.4.2 Plug-on modules



Up to two plug-on modules can be installed on the PCIe-BASE / PCI-BASEII/300/1000/PIO. These modules provide additional channels and are described in the following chapters.

6.4.2.1 MAD12/12a/12b/12f/16/16a/16b/16f

The first analog input channel of a MAD12/12a/12b/12f/16/16a/16b/16f 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 MAD module with the lower address is assigned to the channels 1-16 (channel numbers 0x001 to 0x010), the MAD module with the higher address to channel 17-32 (channel numbers 0x101 to 0x110).

Module	Analog	Channel number	Measuring range	Range
MAD12, MAD16	16 inputs (single-ended)	1..16 (se)	$\pm 1.024\text{V}$	0
	8 inputs (differential)	17..24 (diff)	$\pm 2.048\text{V}$	1
			$\pm 5.120\text{V}$	2
			$\pm 10.240\text{V}$	3
			0.06V..5.06V	4
MAD12a, MAD12f, MAD16a, MAD16f	16 inputs (single-ended)	1..16 (se)	$\pm 1.024\text{V}$	0
	8 inputs (differential)	17..24 (diff)	$\pm 2.048\text{V}$	1
			$\pm 5.120\text{V}$	2
			$\pm 10.240\text{V}$	3
MAD12b, MAD16b	16 inputs (single-ended)	1..16	$\pm 1.024\text{V}$	0
			$\pm 2.048\text{V}$	1
			$\pm 5.120\text{V}$	2
			$\pm 10.240\text{V}$	3

In single-ended mode, 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 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)

```

If the input modules are in differential mode (configured by jumpers), the channel numbers 17..24 must be used.

In differential mode, channels 1 to 8 of the first analog input module are defined by the following constants:

```

C      #define AI1    (AD_CHA_TYPE_ANALOG_IN|0x0011)
         #define AI2    (AD_CHA_TYPE_ANALOG_IN|0x0012)
         ...
         #define AI8    (AD_CHA_TYPE_ANALOG_IN|0x0018)

```

Of course, one input module can be operated in differential and the other in single-ended mode, thus providing for 24 input channels.

6.4.2.2 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 number	Measuring range	Output range
MADDA16, MADDA16n	16 inputs	1..16	0 ($\pm 1.024V$)	0
	2 outputs	1..2	1 ($\pm 2.048V$)	1
			2 ($\pm 5.120V$)	2
			3 ($\pm 10.240V$)	3

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)

```

6.4.2.3 MDA12/12-4/16/16-2i/16-4i/16-8i

Corresponding to the MAD12/12a/12b/12f/16/16a/16b/16f, 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.

Module	Analog	Channel number	Output range	Range
MDA12, MDA16	2 outputs	1..2	±10.24V ±5.12V	0 1
MDA12-4	4 outputs	1..4	±10.24V ±5.12V	0 1
MDA16-2i	2 outputs	1..2	±10.24V	0
MDA16-4i	4 outputs	1..4	±10.24V	0
MDA16-8i	8 outputs	1..8	±10.24V	0

The output ranges of the output modules MDA12/MDA12-4 and MDA16 are configured on the hardware. The user must ensure that the measuring range passed via software complies with the configuration set on the module.

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)
    
```

6.4.2.4 Function generator of the MDA16i-2i/-4i/-8i

The output modules MDA16-2i/-4i/-8i feature a function generator to output periodical signals.

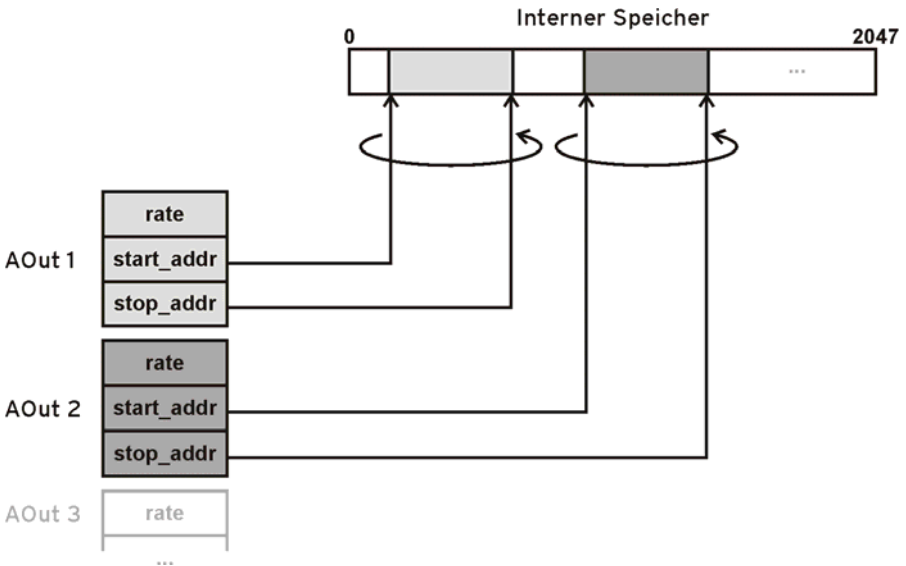


Figure9

The module provides a memory for 2048 measuring values, in which any signal forms can be load. Each output channel of the MDA16-2i/-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 **struct 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 structures of the output controllers to be defined
- **ram**
Memory with output values for the analog output: The analog output is linearly 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 `struct 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 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**
Measuring range number of the output: The measuring range number of the MDA16-2/4/8i with $\pm 10.24V$ 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	<pre>int32_t ad_ioctl (int32_t adh, int32_t ioc, void *par, int32_t size);</pre>
------------------	--

C	<pre>#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; } </pre>
----------	---

```
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));
```

6.5 meM-AD /-ADDA /-ADf / -ADfo



Open the meM-AD/-ADDA/-ADf/-ADfo with the **LIBAD4** by passing the string "**memadusb**" (meM-AD), "**memaddausb**" (meM-ADDA), "**memadfpusb**" (meM-ADf) or "**memadfpusb**" (meM-ADfo) to **ad_open()**. To distinguish between several USB data acquisition systems, the device number is explicitly used (e.g. 1st device with "**memadusb:0**", 2nd device with "**memadusb: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 meM-ADDA devices is removed, the remaining meM-ADDA devices are addressed with "**memaddausb:0**" and "**memaddausb: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 "**memadfpusb:@157**", for example.

6.5.1 Key data and channel numbers meM devices

DAQ system	Analog	Channel number	Input/Output range	Range	Digital	Channel number
meM-AD	16 inputs	1..16	$\pm 5.12V$	0	-	-
meM-ADDA, meM-ADf	16 inputs 1 output	1..16 1	$\pm 5.12V$	0	2 ports (4 bit each)	1: input (bit 0..3) 2: output (bit 0..3)
meM-ADfo	16 inputs 1 output	1..16 1	$\pm 5.12V$	0	2 ports (8 bit each)	1: input (bit 0..3) 2: output (bit 0..3)

The first analog input channel of a meM-AD/-ADDA/-ADf/-ADfo 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 meM-ADDA/-ADf/-ADfo 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 4 (meM-ADfo: 8) lines of the first port (DIO1) are set to input, the 4 (meM-ADfo: 8) 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)
```

6.6 meM-PIO / meM-PIO-OEM



Open the meM-PIO/meM-PIO-OEM with the **LIBAD4** by passing the string **"mempiousb"** to **ad_open()**. To distinguish between several USB data acquisition systems, the device number is explicitly used (e.g. 1st device with **"mempiousb:0"**, 2nd device with **"mempiousb: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 meM-PIO devices is removed, the remaining meM-PIO devices are addressed with **"mempiousb:0"** and **"mempiousb: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 **"mempiousb:@157"**, for example.

6.6.1 Key data and channel numbers meM-PIO(-OEM)

DAQ system	Digital	Channel number
meM-PIO, meM-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. 40).

```
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)
```

6.7 USB-AD / 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.



The USB-AD or USB-PIO/USB-PIO-OEM implements the CDC class as ACM. FreeBSD provides a respective driver for these devices so that the hardware is directly supported by FreeBSD if the umodem driver has been loaded:

```
bash# kldload umodem
bash#
```

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()**. Under FreeBSD, the **LIBAD4** then opens the device **"/dev/cuaU0"** to communicate with the USB-AD or USB-PIO/USB-PIO-OEM. It is the application's responsibility to make sure a USB-AD or USB-PIO/USB-PIO-OEM is registered as **"/dev/cuaU0"**.

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 **LIBAD4** then opens the device `"/dev/cuaU0"` and `"/dev/cuaU1"`.

Besides the automatic allocation of device names, the hardware to be used can also be specified directly under FreeBSD. When calling `ad_open ("usb-ad:/dev/cuaU12"` or `"usb-pio:/dev/cuaU12")`, the **LIBAD4** opens the device `"/dev/cuaU12"`.



The USB-AD or USB-PIO/USB-PIO-OEM implements the CDC class as ACM. Linux provides a respective driver for these devices so that the hardware is directly supported by Linux if the kernel is configured accordingly.

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()`. Under Linux, the **LIBAD4** then opens the device `"/dev/ttyACM0"` to communicate with the USB-AD or USB-PIO/USB-PIO-OEM. It is the application's responsibility to make sure a USB-AD or USB-PIO/USB-PIO-OEM is registered as `"/dev/ttyACM0"`.

Several USB data acquisition systems can be opened by allocating the device name.

When calling `ad_open ("usb-ad:/dev/ttyACM12"` or `"usb-pio:/dev/ttyACM12")`, the **LIBAD4** opens the device `"/dev/ttyACM12"`.

6.7.1 Key data and channel numbers USB-AD

DAQ system	Analog	Channel number	Measuring range	Output range	Digital	Direction
USB-AD	16 inputs 1 output	1..16 1	0 ($\pm 5.12V$)	0 ($\pm 5.12V$)	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 following constants result:

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

6.7.2 Key data and channel numbers USB-PIO(-OEM)

DAQ system	Digital	Channel number
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. 40).

C	<pre>#define DIO1 (AD_CHA_TYPE_DIGITAL_IO 0x0001) #define DIO2 (AD_CHA_TYPE_DIGITAL_IO 0x0002) #define DIO3 (AD_CHA_TYPE_DIGITAL_IO 0x0003)</pre>
---	--

6.8 USB-AD12f



Open the USB-AD12f with the **LIBAD4** by passing the string "**usbad12f**" to **ad_open()**. To distinguish between several USB data acquisition systems, the device number is explicitly used (e.g. 1st device with "**usbad12f:0**", 2nd device with "**usbad12f: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-AD12f devices is removed, the remaining USB-AD12f devices are addressed with "**usbad12f:0**" and "**usbad12f: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 "**usbad12f:@157**", for example.

6.8.1 Key data and channel numbers USB-AD12f

DAQ system	Analog	Channel number	Measuring range	Output range	Digital	Direction
USB-AD12f	16 inputs 1 output	1..16 1	0 ($\pm 10.24V$)	0 ($\pm 5.12V$)	2 ports (4 bit each)	1: input (bit 0..3) 2: output (bit 0..3)

The first analog input channel of a USB-AD12f 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-AD12 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 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      #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)
```

6.9 USB-AD16f



Open the USB-AD16f 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 devices is removed, the remaining USB-AD16f 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.

6.9.1 Key data and channel numbers USB-AD16f

DAQ system	Analog	Channel number	Measuring range	Output range	Digital	Direction
USB-AD16f	16 inputs 2 outputs	1..16 1 .. 2	0 ($\pm 1.024V$) 1 ($\pm 2.048V$) 2 ($\pm 5.12V$) 3 ($\pm 10.24V$)	0 ($\pm 10.24V$)	2 ports (4 bit each)	1: input (bit 0..3) 2: output (bit 0..3)

The 16 analog inputs of a USB-AD16f 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 USB-AD16f are addressed by the following constants:

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

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

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

7 Index

A

ad_analog_in () 37
 ad_analog_out () 37
 ad_calc_run_size () 68
 ad_close () 24
 ad_digital_in () 38
 ad_digital_out () 38
 ad_discrete_in () 26
 ad_discrete_in64 () 27
 ad_discrete_inv () 29
 ad_discrete_out () 30, 31
 ad_discrete_outv () 33
 ad_float_to_sample () 35
 ad_float_to_sample64 () 36
 ad_get_digital_line () 39
 ad_get_drv_version () 41
 ad_get_line_direction () 39
 ad_get_next_run () 70
 ad_get_next_run_f () 70
 ad_get_next_run_f64 () 71
 ad_get_range_count () 24
 ad_get_range_info () 25
 ad_get_sample_layout () 64
 ad_get_samples () 65
 ad_get_samples_f () 66
 ad_get_samples_f64 () 67
 ad_get_version () 40
 ad_open () 20, 22
 ad_poll_scan_state () 71
 ad_sample_to_float () 33
 ad_sample_to_float64 () 34
 ad_set_digital_line () 38
 ad_set_line_direction () 40
 ad_start_mem_scan () 62
 ad_start_scan () 63
 ad_stop_scan () 72
 Analog output
 Set 30, 31
 Set several 33

B

Buffer 20, 42, 47, 51, 69
 buffer_start 64
 bytes_per_run 47, 69

C

Case sensitivity 20, 22
 cha 43, 80, 86, 93, 94
 chac 93
 Channel number 26, 27, 29, 30, 31, 33,
 42, 43, 73
 Channel type 73
 chav 93
 Continuous scan 47, 54
 Conversion
 Measuring value into voltage value 33, 34
 Voltage value into measuring value 35, 36
 Copyright 10
 Counter
 Configuration 79, 84
 Counter channel 80, 86

D

Data acquisition system
 Close 19, 24
 Name 20
 Open 19, 22
 Open several different ones 20, 22
 Open several equal ones 22
 Differential 89
 Digital channel
 Get direction 39
 Set direction 40
 Direction 39
 Driver version 41

E

Error number 20, 22, 72

F

flags 47, 48, 81, 87
 FreeBSD 7, 14
 Function generator 92

G

GetLastError 20, 22

H

Header file 19

I

iM-3250 74
 iM-3250T 74
 iM-AD25 74
 iM-AD25a 74
 Input
 Direction 40
 Input line 39
 Inputs
 Order 63, 64
 Installation
 FreeBSD 14
 Linux 16
 Mac OS X 11
 Windows® 11
 Internet address 9

L

LAN-AD16f 77
 Channel number 77
 Counter 78
 Digital ports 78
 Linux 7, 16

M

Mac OS X 7, 11
 MAD12 88
 MAD12a 88
 MAD12b 88
 MAD12f 88

MAD16 88
 MAD16a 88
 MAD16b 88
 MAD16f 88
 MADDA16 90
 MADDA16n 90
 Maximum 44
 MDA12 91
 MDA12-4 91
 MDA16 91
 MDA16-2i 91
 MDA16-4i 91
 MDA16-8i 91
 Mean value 44
 Measuring range 26, 27, 29, 43, 73
 Information 25
 Middle 27, 28
 Number 24
 Measuring value 30, 31, 33, 34, 35, 36,
 70
 Read out 60
 meM devices
 Digital ports 99
 Order 97, 99
 Serial number 97, 99
 meM-AD 97
 meM-ADDA 97
 meM-ADf 97
 meM-ADfo 97
 Memory management of the measuring
 values
 Internal 60, 71
 Memory-only scan 49, 62, 70, 71
 meM-PIO 99
 meM-PIO-OEM 99
 Minimum 44
 mode 80, 86
 mux_a 81, 86
 mux_b 81, 86
 mux_rst 81, 86

N

Name 20
 Network byte order 70
 Number of measuring values 42, 44, 47,
 48, 54

O

OR operator (!) 73

Output

Direction 40

Set 30, 31

Set several 33

Output line 39

Output range 30, 31, 33, 73

Overrun of the samples 42, 54, 72

P**PCI cards**

Serial number 82

PCI-BASE1000 82

Digital ports 83

PCI-BASE300 82

Digital ports 83

PCI-BASEII 82

Digital ports 83

PCIe cards

Serial number 82

PCIe-BASE 82

Digital ports 82

PCI-PIO 82

Digital ports 83

Pointer 51, 62

posthist 47, 48

posthist_samples 65

Posthistory 47, 60

Number of measuring values 65

prehist 47

prehist_samples 64

Prehistory 47, 60

First measuring value 64

Number of measuring values 65

R

ram 93

range 43, 94

Range limit 26, 70

rate 94

ratio 43

Reading out measuring values 51

Result 72

RMS 43, 44

Root mean square value 43

RUN 49, 54

runs_pending 48

S

Sample rate 58

sample_rate 47, 68

samples_per_run 44, 47, 69

Sampling period 42, 68

Sampling pulse 59

Sampling rate 45, 47, 54

Scan 20, 42

Continuous 42, 54

First measuring value 64

Memory-only 42

Parameters 42

Start 50

State 48, 60, 71

Stop 53

With trigger 60

Serial number 82, 97, 99, 100, 103, 105

Single value

Read 26, 27

Read several 29

Single-ended 89

start 64

Start scan 50

start_addr 94

Stop scan 53, 72

stop_addr 94

Storage

Interval 43

Ratio 58

Type 43, 44

store 43

struct ad_scan_cha_desc 43

struct ad_scan_desc 46

struct ad_scan_state 48

T

ticks_per_run 47, 69

trg_mode 44

trg_par 44

Trigger 44, 45, 47, 48, 60

Condition 45, 60

Negative Edge 46

- Parameters 44
- Positive Edge 46
- Settings 42
- Window 46

U

- USB-AD 100
 - Channel number 102
 - Digital ports 102
 - Serial number 100
- USB-AD12f 103
 - Channel number 104
 - Digital ports 104
 - Order 103
 - Serial number 103
- USB-AD16f 105
 - Channel number 105
 - Digital ports 106
 - Order 105
 - Serial number 105
- USB-PIO 100
 - Channel number 103
 - Digital ports 103
 - Direction 103

- Serial number 100
- USB-PIO-OEM 100
 - Channel number 103
 - Digital ports 103
 - Direction 103
 - Serial number 100

V

- Version
 - Driver 41
 - LIBAD4.DLL 40
- Voltage value 33, 34, 35, 36, 51

W

- Window trigger 46
- Windows® 7, 11

Z

- zero 43
- Zero level 43