LIBAD4

Bibliothek für Programmierschnittstelle LIBAD4

Programmierhandbuch

Version 5.0 (64bit)

© BMC Messsysteme GmbH

Inhaltsverzeichnis

1 Überblick 1
1.1 Einleitung 1
1.2 BMC Messsysteme GmbH 1
1.3 Urheberrecht 2
2 Installation 3
2.1 Installation 3
2.2 Weitergabe der Bibliothek 3
3 Grundlagen 4
3.1 Einführung 4
4 Einzelwerterfassung 6
4.1 Funktionsbeschreibung (Einzelwerte) 6
4.1.1 ad_open 6
4.1.2 ad_close 8
4.1.3 ad_get_range_count 8
4.1.4 ad_get_range_info 9
4.1.5 ad_discrete_in 9
4.1.6 ad_discrete_in64 11
4.1.7 ad_discrete_inv 11
4.1.8 ad_discrete_out 12
4.1.9 ad_discrete_out64 13
4.1.10 ad_discrete_outv 14
4.1.11 ad_sample_to_float 16
4.1.12 ad_sample_to_float64 16
4.1.13 ad_float_to_sample 17
4.1.14 ad_float_to_sample64 18
4.1.15 ad_analog_in 18
4.1.16 ad_analog_out 19
4.1.17 ad_digital_in 19
4.1.18 ad_digital_out 19
4.1.19 ad_set_digital_line 19
4.1.20 ad_get_digital_line 20
4.1.21 ad_get_line_direction 20
4.1.22 ad_set_line_direction 20
4.1.23 ad_get_version 20
4.1.24 ad_get_drv_version 21
4.1.25 ad_get_product_info 21
5 Scanvorgang 22
5.1 Einführung 22
5.2 Scanparameter 22

5.2.1 struct ad_scan_cha_desc ... 22
5.2.1.1 Speicherarten ... 23
5.2.1.2 Triggerarten ... 24
5.2.2 struct ad_scan_desc ... 25
5.2.3 struct ad_scan_state ... 26
5.2.4 struct ad_scan_pos ... 27
5.2.5 struct ad_cha_layout ... 27
5.3 Memory-only Scan ... 27
5.3.1 Starten eines Scans ... 28
5.3.2 Auslesen der Messwerte ... 28
5.3.3 Stoppen des Scans ... 29
5.4 Kontinuierliche Messung ... 30
5.4.1 Aufbau eines RUNs ... 30
5.4.2 Ein Messwert pro RUN ... 33
5.4.3 Signale mit unterschiedlicher Speicherrate ... 34
5.5 Messung mit Triggerung ... 34
5.5.1 Parameter der Scankanäle bei Triggerung ... 35
5.6 Funktionsbeschreibung (Scan) ... 38
5.6.1 ad_start_mem_scan ... 38
5.6.2 ad_start_scan ... 40
5.6.3 ad_get_sample_layout ... 40
5.6.4 ad_get_samples ... 41
5.6.5 ad_get_samples_f ... 41
5.6.6 ad_get_samples_f64 ... 42
5.6.7 ad_calc_run_size ... 43
5.6.8 ad_get_next_run ... 43
5.6.9 ad_get_next_run_f ... 44
5.6.10 ad_get_next_run_f64 ... 44
5.6.11 ad_poll_scan_state ... 45
5.6.12 ad_stop_scan ... 45
6 Messsysteme ... 46
6.1 Hinweise ... 46
6.2 LAN-AD16fx / AMS42/84-LAN16fx ... 46
6.2.1 Kanalnummern LAN-AD16fx / AMS42/84-LAN16fx ... 46
6.2.2 Konfiguration der LAN-AD16fx / AMS42/84-LAN16fx Zähler ... 48
6.3 PCIe-BASE / PCI-BASEII / PCI-PIO ... 50
6.3.1 Digitalports und Zähler ... 51
6.3.1.1 PCIe-BASE / PCI-BASEII / PCI-PIO ... 51
6.3.1.2 Konfiguration der Zähler ... 52
6.3.2 Aufsteckmodule ... 54
6.3.2.1 MADDA16/16n ... 55
6.3.2.2 MDA16-4i/-8i ... 55
6.3.2.3 Funktionsgenerator der MDA16-4i/-8i ... 56

 © BMC Messsysteme GmbH

6.4 USB-AD / USB-AD-OEM / USB-PIO / USB-PIO-OEM 60
6.4.1 Eckdaten und Kanalnummern USB-AD / USB-AD-OEM 60
6.4.2 Eckdaten und Kanalnummern USB-PIO(-OEM) 61
6.5 USB-AD14f 61
6.5.1 Eckdaten und Kanalnummern USB-AD14f 61
6.6 USB-AD16f / AMS42/84-USB 62
6.6.1 Eckdaten und Kanalnummern USB-AD16f / AMS42/84-USB 62
6.7 USB-OI16 63
6.7.1 Eckdaten und Kanalnummern USB-OI16 63
6.7.2 Kanalnummern USB-OI16 64
6.7.3 Konfiguration der USB-OI16 Zähler 64

7 Index 68

1 Überblick

1.1 Einleitung

Die Bibliothek LIBAD4 ist eine Schnittstelle zu allen **Messsysteme**n der BMC Messsysteme GmbH. Diese Schnittstelle erlaubt zum Beispiel das Lesen und Schreiben von Einzelwerten, wie das Einlesen eines Analogeingangs oder das Ausgeben eines Werts an einen Analogausgang.

Neben der Ein-/Ausgabe von Einzelwerten kann mit der LIBAD4 eine Messung durchgeführt werden. Dieser Scan der Eingangskanäle findet im entsprechenden Treiber statt und ist deshalb zeitlich von der Applikation entkoppelt. Damit ist es möglich, schnell und ohne Verlust von Messwerten die Eingangskanäle abzutasten.

- **LIBAD4 ist eine 64-Bit Schnittstelle.**
- **Bitte beachten Sie, dass alle Beispielcodes in diesem Handbuch aus Gründen der Einfachheit bewusst auf eine Fehlerbehandlung verzichten. Selbstverständlich muss diese in selbst geschriebenen Programmen realisiert werden.**

1.2 BMC Messsysteme GmbH

BMC Messsysteme GmbH steht für innovative Messtechnik "made in Germany". Vom Sensor bis zur Software bieten wir alle für die Messkette benötigten Komponenten an.

Unsere Hard- und Software ist aufeinander abgestimmt und dadurch besonders anwenderfreundlich. Darüber hinaus legen wir größten Wert auf die Einhaltung gängiger Industriestandards, die das Zusammenspiel vieler Komponenten erleichtern.

BMC Messsysteme Produkte finden Sie im industriellen Großeinsatz ebenso wie in Forschung und Entwicklung oder im privaten Anwenderbereich. Wir fertigen unter Einhaltung der ISO-9000-Vorschriften, denn Standards und Zuverlässigkeit sind uns wichtig - für Sie und für uns!

Neueste Informationen finden Sie im Internet auf unserer Homepage unter http://www.bmcm.de.

© BMC Messsysteme GmbH

1.3 Urheberrecht

Die Programmierschnittstelle **LIBAD4** mit allen Erweiterungen wurde mit größtmöglicher Sorgfalt erstellt und geprüft. Die BMC Messsysteme GmbH gibt keine Garantien, weder in Bezug auf dieses Handbuch noch in Bezug auf die in diesem Buch beschriebene Hard- und Software, ihre Qualität, Durchführbarkeit oder Verwendbarkeit für einen bestimmten Zweck. Die BMC Messsysteme GmbH haftet in keinem Fall für direkt oder indirekt verursachte oder erfolgte Schäden, die entweder aus unsachgemäßer Bedienung oder aus irgendwelchen Fehlern am System resultieren. Änderungen, die dem technischen Fortschritt dienen, bleiben uns vorbehalten.

Die Programmierschnittstelle **LIBAD4** sowie das vorliegende Handbuch und sämtliche darin verwendeten Namen, Marken, Bilder und sonstige Bezeichnungen und Symbole sind ihrerseits gesetzlich sowie aufgrund nationaler und internationaler Verträge geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, der Entnahme von Abbildungen, der Funksendung, der Wiedergabe auf photomechanischem oder ähnlichem Wege bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Die Reproduktion der Programme und des Programmhandbuchs sowie die Weitergabe an Dritte ist nicht gestattet. Ihre rechtswidrige Verwendung oder sonstige rechtliche Beeinträchtigung wird straf- und zivilrechtlich verfolgt und kann zu empfindlichen Sanktionen führen.

Copyright © 2014

Stand: 24. September 2019

BMC Messsysteme GmbH

Hauptstraße 21
82216 Maisach
DEUTSCHLAND

Tel.: +49 8141/404180-1
Fax: +49 8141/404180-9
E-Mail: info@bmcm.de

2 Installation

2.1 Installation

Die **LIBAD4** ist als "dynamic link library" realisiert. Das Installationsprogramm kopiert die Bibliothek inklusive aller Headerfiles und den Beispielprogrammen auf die Festplatte.

Damit Programme auf die **`libad4.dll`** zugreifen können, sollte diese in das entsprechende Programmverzeichnis kopiert werden.

Alle Funktionen der LIBAD4 verwenden die Aufrufkonventionen cdecl von C. Soll die Bibliothek unter einer anderen Programmiersprache als C/C++ verwendet werden, muss sichergestellt werden, dass diese die Aufrufkonventionen von C für die LIBAD4 Funktionen verwendet.

2.2 Weitergabe der Bibliothek

Damit eine Applikation auf die Funktionen der **LIBAD4** Bibliothek zugreifen kann, muss jene auf dem Zielsystem installiert werden. Aus diesem Grund ist die Weitergabe der folgenden Files ausdrücklich erlaubt (die Versionsnummer Ihrer **LIBAD4** ist eventuell eine andere und muss entsprechend angepasst werden).

`libad4.dll`

Es ist Aufgabe des Installationsprogramms der erstellten Applikation, das entsprechende File zusammen mit der Applikation zu installieren. Neben der DLL muss auch die entsprechende Laufzeitbibliothek des verwendeten C Compilers installiert werden. Die LIBAD4 benötigt die entsprechende Version der MSVCRT (MSVCRT110.DLL), die auch von der Applikation installiert werden muss (z.B. durch Installation von Microsoft Visual C++ 2012 Redistributable). Auf keinen Fall sollte das LIBAD4 SDK verwendet werden, um die LIBAD4 Bibliothek auf dem Zielrechner zu installieren.

Bitte beachten Sie, dass alle anderen Files aus dem LIBAD4 SDK nicht weitergegeben werden dürfen!

© BMC Messsysteme GmbH

3 Grundlagen

3.1 Einführung

Die von **LIBAD4** exportierten Funktionen und die verwendeten Konstanten werden einem C/C++ Programm in der Headerdatei **`libad.h`** zur Verfügung gestellt.

Die exakten Definitionen der in diesem Handbuch beschriebenen C/C++ Aufrufe und Strukturen sind in den aktuellen Headerdateien definiert.

Die **LIBAD4** stellt zwei Funktionen zur Verfügung, mit denen ein Messsystem geöffnet bzw. wieder geschlossen werden kann. Mit der Funktion ad_open**()** wird ein Messsystem geöffnet, mit ad_close**()** wieder geschlossen. Folgendes Beispiel demonstriert das prinzipielle Vorgehen:

Prototyp

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

C

```
#include "libad.h"

...
int32_t adh;
...

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

Der Funktion ad_open**()** wird der Name des Messsystems übergeben. Der übergebene String wird ohne Berücksichtigung von Groß- und Kleinschreibung verwendet, d. h. "usb-ad" und "USB-AD" öffnen beide das USB-AD. Als Rückgabewert liefert die Funktion einen Handle, der in allen weiteren Aufrufen an die LIBAD4 benötigt wird. Im Fehlerfall wird **–1** zurückgegeben. Die Fehlernummer lässt sich unter Windows® mit **GetLastError()** abfragen. Es werden nur Fehlnernummern des entsprechenden Betriebssystem zurückgeliefert. Eine Liste der Fehlermeldungen findet sich unter Windows® beispielsweise in der Datei winerror.h aus dem entsprechenden SDK.

Es ist durchaus auch möglich, mehrere Messsysteme gleichzeitig zu öffnen, ad_open**()** gibt dann für jeden geöffneten Treiber einen anderen Handle zurück. Genaue Hinweise dazu entnehmen Sie bitte der Beschreibung der Funktion ad_open**()**, Seite 6.

Die unterstützten Messsysteme sind im Kapitel "Messsysteme" ab Seite 46 beschrieben. Dort sind auch die benötigten Kanalnummern für die Ein- und Ausgangkanäle und die entsprechenden Messbereiche definiert.

© BMC Messsysteme GmbH

Sobald ein Messsystem geöffnet worden ist, lassen sich Messwerte von den Eingängen einlesen (siehe "ad_discrete_in" , S. 9) oder die Werte für die Ausgänge festlegen (siehe "ad_discrete_out" , S. 12). Genaue Hinweise dazu entnehmen Sie bitte dem Kapitel "Einzelwerterfassung", Seite 6.

Neben der Einzelwertabfrage von Messwerten kann die LIBAD4 auch einen Scanvorgang starten. Dieser tastet mehrere Eingangskanäle in einem festen Zeitraster ab und liefert die erfassten Messwerte in einem Buffer zurück. Das Programmieren eines Scans ist im Kapitel "Scanvorgang" ab Seite 22 beschrieben.

Die Abfrage von Einzelwerten führt dazu, dass pro Abfrage ein Befehl zum Messsystem geschickt werden muss. Bei der Programmierung eines Scans wird nur beim Start der Messung ein Befehl an das Gerät geschickt. Danach sendet das Messsystem kontinuierlich Messdaten.

Nachdem die Befehlsübertragung immer mit einer gewissen Latenzzeit behaftet ist, kann aus diesem Grund die Abfrage von Einzelwerten niemals in der Geschwindigkeit durchgeführt werden, die bei der Programmierung eines Scans erreicht wird.

© BMC Messsysteme GmbH

4 Einzelwerterfassung

4.1 Funktionsbeschreibung (Einzelwerte)

Alle Funktionen der LIBAD4 sind thread-safe, solange dies in der Funktionsbeschreibung nicht ausdrücklich anders spezifiziert ist.

4.1.1 ad_open

Prototype

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

C

```
#include "libad.h"

...
int32_t adh;
...

  adh = ad_open ("usb-ad");

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

...

  ad_close (adh);
```

Die Funktion **ad_open()** stellt eine Verbindung zum Messsystem her. Es wird der Name des Messsystems übergeben. Der übergebene String wird ohne Berücksichtigung von Groß- und Kleinschreibung verwendet, d.h. "**pcibase**" und "**PCIbase**" öffnen beide eine PCIe-BASE / PCI-BASEII / PCI-PIO. Als Rückgabewert liefert die Funktion einen Handle, der in allen weiteren Aufrufen an die LIBAD4 benötigt wird. Im Fehlerfall wird **–1** zurückgegeben. Die Fehlernummer lässt sich unter Windows® mit **GetLastError()**erfragen.

Es ist durchaus möglich, mehrere (verschiedene) Messsysteme zu öffnen, **ad_open()** gibt dann für jeden geöffneten Treiber einen anderen Handle zurück.

Gerätespezifische Informationen (z. B. Name des Messsystems) zu den unterstützten Messsystemen sind in den gleichnamigen Kapiteln enthalten:

LAN-AD16fx / AMS42/84-LAN

PCIe-BASE / PCI-BASEII / PCI-PIO

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

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

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

Folgendes Beispiel öffnet ein USB-AD und eine USB-PIO:

C

```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);
```

Sollen mehrere Messsysteme gleichen Typs geöffnet werden, dann ist die Nummer des Messsystems mit Doppelpunkt getrennt an den Namen anzuhängen. Folgendes Beispiel öffnet zwei USB-AD Geräte:

C

```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);
```

Alternativ lässt sich ein Messsystem über seine Seriennummer öffnen. Dabei ist die Seriennummer mit einem @ Zeichen nach dem Doppelpunkt anzugeben.

Das folgende Beispiel öffnet die zwei USB-AD Geräte mit den Seriennummern 157 und 158.

 © BMC Messsysteme GmbH

C

```
#include "libad.h"

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

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

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

4.1.2 ad_close

Prototype

```
int32_t
ad_close (int32_t adh);

C#include "libad.h"

...
int32_t adh;
...

  adh = ad_open ("usb-ad");

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

  ad_close (adh);
```

Die Funktion **ad_close()** schließt ein Messsystem wieder. Der Rückgabewert der Funktion ist **0** oder im Fehlerfall die entsprechende Fehlernummer.

4.1.3 ad_get_range_count

Prototype

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

Die Funktion **ad_get_range_count()** liefert die Anzahl der Messbereiche des Kanals **cha** zurück.

4.1.4 ad_get_range_info

Prototype

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

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

C

```
#include "libad.h"

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

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

Die Funktion **ad_get_range_info()** liefert die Messbereichsinformation des Messbereichs **range** des Kanals **cha** zurück.

4.1.5 ad_discrete_in

Prototype

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


© BMC Messsysteme GmbH

C

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

...

  adh = ad_open ("usb-ad");

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

Die Funktion **ad_discrete_in()** liefert einen Einzelwert des angegebenen Kanals. Dabei wird im Parameter **cha** neben der eigentlichen Kanalnummer noch der Kanaltyp angegeben:

AD_CHA_TYPE_ANALOG_IN	für Analogeingänge
AD_CHA_TYPE_ANALOG_OUT	für Analogausgänge
AD_CHA_TYPE_DIGITAL_IO	für Digitalkanäle
AD_CHA_TYPE_COUNTER	für Zählerkanäle

Je nach Messsystem stehen unterschiedliche Kanäle zur Verfügung. Diese sind im Kapitel "Messsysteme" (siehe S. 46) dokumentiert. Neben der Kanalnummer wird der Funktion noch der Messbereich übergeben, in dem der Eingangskanal abgetastet werden soll. Der Messbereich wird für digitale Kanäle ignoriert.

Die Funktion **ad_discrete_in()** liefert für analoge Kanäle in ***data** einen Wert zwischen **0x00000000** und **0xffffffff** zurück. Dabei entspricht der Wert **0x00000000** der unteren Messbereichsgrenze, der Wert **0x100000000** der oberen Messbereichsgrenze (dieser Wert wird bei 32 Bit nicht erreicht, und daher maximal **0xffffffff** zurückgegeben). Der Wert **0x80000000** entspricht der Messbereichsmitte, bei einem symmetrischen, bipolaren Eingang also 0.0V.

Für die Umrechnung eines solchen Werts in einen Spannungswert steht die Funktion ad_sample_to_float**()** zur Verfügung. Die Hilfsfunktion ad_analog_in**()** übergibt den Messwert direkt als Spannung.

Die Kanalnummer und die Nummer des Messbereichs sind abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (siehe "Messsysteme", S. 46).

4.1.6 ad_discrete_in64

Prototype

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

Cint32_t adh;
int32_t st;
uint64_t data;

...

  adh = ad_open ("usb-ad");

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

...

  ad_close (adh);
```

Die Funktion **ad_discrete_in64()** liefert einen Einzelwert des angegebenen Kanals. Neben der Kanalnummer wird der Funktion noch der Messbereich übergeben, in dem der Eingangskanal abgetastet werden soll. Der Messbereich wird für digitale Kanäle ignoriert.

Die Funktion **ad_discrete_in64()** liefert einen Wert zwischen **0x0000000000000000** (der unteren Messbereichsgrenze) und **0x10000000000000000** (der oberen Messbereichsgrenze). Die vollen 64 Bit werden nur von speziellen 64-Bit Messsystemen benutzt. Der Wert **0x8000000000000000** entspricht der Messbereichsmitte, bei einem symmetrischen, bipolaren Eingang also 0.0V.

Für die Umrechnung eines solchen Werts in einen Spannungswert steht die Funktion ad_sample_to_float**()** zur Verfügung. Die Hilfsfunktion ad_analog_in**()** übergibt den Messwert direkt als Spannung.

Die Kanalnummer und die Nummer des Messbereichs sind abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (siehe "Messsysteme", S. 46).

4.1.7 ad_discrete_inv

Prototype

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


© BMC Messsysteme GmbH

C

```
#define CHAC 3

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

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

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

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

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

Die Funktion **ad_discrete_inv()**liest **chac** Eingänge auf einmal. Dabei können analoge und digitale Eingänge gemischt werden. Neben den Kanalnummern werden der Funktion noch die Messbereiche übergeben, in denen die Eingangskanäle betrieben werden.

Im Normalfall wird die Routine **ad_discrete_inv()** etwas schneller abgearbeitet, als der mehrmalige Aufruf der Funktion ad_discrete_in64**()** in einer entsprechenden Schleife.

Im Gegensatz zu ad_discrete_in**()** und ad_discrete_in64**()** werden an **ad_discrete_inv()** Kanalnummern, Messbereiche und Wertvariablen in Feldern übergeben. Die Feldwerte werden dabei analog zu ad_discrete_in64**()** gesetzt.

4.1.8 ad_discrete_out

Prototype

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

C

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

  adh = ad_open ("usb-ad");

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

  ad_close (adh);
```

Die Funktion **ad_discrete_out()** setzt einen Ausgang. Neben der Kanalnummer wird der Funktion noch ein Ausgangsbereich übergeben, in dem der Ausgangskanal betrieben wird (nur bei Messsystemen, die den Ausgangsbereich softwaremäßig umschalten können). Andernfalls ist softwaremäßig dafür Sorge zu tragen, dass der angegebene Ausgangsbereich mit den Hardwareeinstellungen übereinstimmt.

Wie beim analogen Eingangskanal entspricht der Wert **0x00000000** eines Analogausgangs der niedrigsten Ausgangsspannung, der Wert **0x100000000** der höchsten Ausgangsspannung (da **0x100000000** von 32 Bit nicht erreicht wird, kann **ad_discrete_out()** maximal **0xffffffff** übergeben werden).

Mittels ad_float_to_sample**()** lässt sich ein Spannungswert (float) in einen Digitalwert zur Übergabe an **ad_discrete_out()** umrechnen. Die Hilfsfunktion ad_analog_out**()** übergibt den Messwert direkt als Spannung.

Die Kanalnummer und die Nummer des Ausgangsbereichs sind abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (siehe "Messsysteme", S. 46).

4.1.9 ad_discrete_out64

Prototype

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


© BMC Messsysteme GmbH

C

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

...

  adh = ad_open ("pcibase");

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

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

  ad_close (adh);
```

Die Funktion **ad_discrete_out64()** setzt einen Ausgang. Neben der Kanalnummer wird der Funktion noch ein Ausgangsbereich übergeben, in dem der Ausgangskanal betrieben wird (nur bei Messsystemen, die den Ausgangsbereich softwaremäßig umschalten können). Andernfalls ist softwaremäßig dafür Sorge zu tragen, dass der angegebene Ausgangsbereich mit den Hardwareeinstellungen übereinstimmt.

Wie beim analogen Eingangskanal entspricht der Wert **0x0000000000000000** eines Analogausgangs der niedrigsten Ausgangsspannung, der Wert **0x10000000000000000** der höchsten Ausgangsspannung. Die vollen 64 Bit von **ad_discrete_out64()** werden wie bei den Eingängen nur von speziellen 64-Bit Messsystemen genutzt.

Zur Verfügung steht für die Umrechnung eines Spannungswerts (float) in einen Digitalwert zur Ausgabe mittels **ad_discrete_out64()** die Funktion ad_float_to_sample64**()**. Die Hilfsfunktion ad_analog_out**()** übergibt den Messwert direkt als Spannung.

Die Kanalnummer und die Nummer des Ausgangsbereichs sind abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (siehe "Messsysteme", S. 46).

4.1.10 ad_discrete_outv

Prototype

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

C

```
#define CHAC 3

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

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

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

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

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

Die Funktion **ad_discrete_outv()**setzt **chac** Ausgänge auf einmal. Dabei können analoge und digitale Ausgänge gemischt werden. Neben den Kanalnummern werden der Funktion noch die Ausgangsbereiche übergeben, in denen die Ausgangskanäle betrieben werden

Im Normalfall wird die Routine **ad_discrete_outv()** etwas schneller abgearbeitet, als der mehrmalige Aufruf der Funktion ad_discrete_out64**()** in einer entsprechenden Schleife.

Im Gegensatz zu ad_discrete_out**()** und ad_discrete_out64**()** übergibt man an **ad_discrete_outv()** Kanalnummern, Ausgangsbereiche und Werte in Feldern. Die Feldwerte müssen wie in ad_discrete_out64**()** gesetzt werden.

© BMC Messsysteme GmbH

4.1.11 ad_sample_to_float

Prototype

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

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

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

Rechnet einen Messwert in den entsprechenden Spannungswert um.

Die Kanalnummer und die Nummer des Messbereichs sind abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (siehe "Messsysteme", S. 46).

4.1.12 ad_sample_to_float64

Prototype

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

C

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

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

Rechnet einen Messwert in den entsprechenden Spannungswert um.

Die Kanalnummer und die Nummer des Messbereichs sind abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (siehe "Messsysteme", S. 46).

4.1.13 ad_float_to_sample

Prototype

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

C

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

  adh = ad_open ("usb-ad");

...

  cha = AD_CHA_TYPE_ANALOG_OUT|1;
  range = 0;

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

  ad_close (adh);
```

 © BMC Messsysteme GmbH

Rechnet einen Messwert in den entsprechenden Spannungswert um.

Die Kanalnummer und die Nummer des Messbereichs sind abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (siehe "Messsysteme", S. 46).

4.1.14 ad_float_to_sample64

Prototype

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

C

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

...

  adh = ad_open ("usb-ad");

...

  cha = AD_CHA_TYPE_ANALOG_OUT|1;
  range = 0;

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

...

  ad_close (adh);
```

Rechnet einen Spannungswert in den entsprechenden Messwert um.

Die Kanalnummer und die Nummer des Messbereichs sind abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (siehe "Messsysteme", S. 46).

4.1.15 ad_analog_in

Prototype

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

Diese Hilfsfunktion ruft ad_discrete_in**()** auf und rechnet dann den gemessenen Wert mit ad_sample_to_float**()** in den Spannungswert um. Dabei werden nur analoge Eingänge unterstützt, d. h. intern wird als Kanalnummer **AD_CHA_TYPE_ANALOG_IN|cha** verwendet.

Die Kanalnummer und die Nummer des Messbereichs sind abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (siehe "Messsysteme", S. 46).

4.1.16 ad_analog_out

Prototype

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

Diese Hilfsfunktion rechnet den Spannungswert mit ad_float_to_sample**()** um und ruft dann ad_discrete_out**()** auf. Dabei werden nur analoge Ausgänge unterstützt, d. h. intern wird **AD_CHA_TYPE_ANALOG_OUT|cha** als Kanalnummer verwendet.

Die Kanalnummer und die Nummer des Messbereichs sind abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (siehe "Messsysteme", S. 46).

4.1.17 ad_digital_in

Prototype

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

Diese Hilfsfunktion ruft ad_discrete_in**()** mit der Kanalnummer **AD_CHA_TYPE_DIGITAL_IO|cha** auf.

4.1.18 ad_digital_out

Prototype

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

Diese Hilfsfunktion ruft ad_discrete_out**()** mit der Kanalnummer **AD_CHA_TYPE_DIGITAL_IO|cha** auf.

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);
```

Diese Hilfsfunktion liest den Kanal **AD_CHA_TYPE_DIGITAL_IO|cha** und setzt dann entsprechend dem Parameter **flag** die Leitung mit der Nummer **line**. Ist **flag** gleich 0, wird die Leitung zurückgesetzt, ist **flag** ungleich 0, wird die Leitung gesetzt. Die erste Leitung in einem Digitalkanal hat die Nummer 0.

© BMC Messsysteme GmbH

4.1.20 ad_get_digital_line

Prototype

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

Diese Hilfsfunktion liest den Kanal **AD_CHA_TYPE_DIGITAL_IO|cha** und setzt dann **flag** entsprechend der Leitung **line**. Ist die Leitung low, wird **flag** auf 0 gesetzt, ansonsten auf 1. Die erste Leitung eines Digitalkanals hat die Nummer 0.

4.1.21 ad_get_line_direction

Prototype

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

Liefert eine Bitmaske, die die Richtung der Digitalleitung beschreibt. Jedes gesetzte Bit definiert eine Eingangsleitung, jedes gelöschte Bit eine Ausgangsleitung. Das Bit #0 legt die Richtung der ersten Leitung des Digitalports fest.

4.1.22 ad_set_line_direction

Prototype

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

Setzt die Ein-/Ausgaberichtung aller Leitungen eines Digitalkanals **cha**. Dazu wird eine Bitmaske übergeben, die die Richtung der Leitung des Digitalkanals beschreibt. Jedes gesetzte Bit definiert eine Eingangsleitung, jedes gelöschte Bit eine Ausgangsleitung. Das Bit #0 legt die Richtung der 1. Leitung des Digitalports fest.

Beispielsweise werden mit **0xFFFF** alle Digitalleitungen auf Eingang gesetzt, mit **0x0000** auf Ausgang.

Je nach Messsystem kann eventuell nicht jede Leitung ad_open einzeln in der Richtung umgeschaltet werden oder die Richtung ist fest eingestellt (z. B. der Digitalport des USB-AD14f).

4.1.23 ad_get_version

Prototype

```
uint32_t
ad_get_version ();
```

Liefert die Version der LIBAD4.DLL zurück. Diese ID lässt sich mit den Makros **AD_MAJOR_VERS()**, **AD_MINOR_VERS()** und **AD_BUILD_VERS()** zerlegen.

4.1.24 ad_get_drv_version

Prototype

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

Liefert die Version des Messkartentreibers zurück, auf den die LIBAD4 aufsetzt.

4.1.25 ad_get_product_info

Prototype

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

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

Mit der Funktion **ad_get_product_info()** wird die Seriennummer, Firmwareversion und der Produktname des mit **ad_open()** geöffneten Messsystems abgefragt.

Bei dem Parameter id = 0 wird die Information des geöffneten Messsystems geliefert. Mit id = 1 oder 2 kann, falls vorhanden, die Produktinformation eines Messmoduls in dem Messsystem abgefragt werden (z. B. MADDA16 mit PCIe-BASE).

© BMC Messsysteme GmbH

5 Scanvorgang

5.1 Einführung

Neben der Einzelwertabfrage von Messwerten kann die LIBAD4 auch einen Scanvorgang starten. Dieser tastet mehrere Eingangskanäle in einem festen Zeitraster ab und liefert die erfassten Messwerte in einem Buffer zurück.

Dabei unterscheidet die LIBAD4 zwischen so genannten "memory-only"-Messungen und kontinuierlichen Messungen. Eine "memory-only"-Messung ist so kurz, dass die gesamten Messdaten des Scans im Hauptspeicher des PCs untergebracht werden können. Dazu wird der Scanvorgang eingestellt, gestartet und mit dem Ende des Scans liegen alle Messwerte in einem Buffer vor.

Eine kontinuierliche Messung liefert während des Scanvorgangs die erfassten Messwerte an den Aufrufer ab. Dabei kann für Messsysteme, die selbstständig einen Scanvorgang durchführen (z. B. LAN-AD16fx / AMS42/84-LAN, PCIe-BASE / PCI-BASEII / PCI-PIO mit MDA16-4i/-8i/MADDA16/16n, USB-AD16f / AMS42/84-USB, USB-AD14f) eine interne Messdaten-Speicherverwaltung aktiviert werden. Alternativ dazu lässt sich auch eine eigene Speicherverwaltung realisieren. Der Aufrufer ist in beiden Fällen dafür verantwortlich, die Messdaten schnell genug aus der LIBAD4 auszulesen und zu speichern – andernfalls kommt es zu einem Überlauf der Messwerte und der Scanvorgang wird abgebrochen.

5.2 Scanparameter

Der Scanvorgang wird mit Hilfe der zwei Strukturen **struct ad_scan_desc** und **struct ad_scan_cha_desc** definiert. In **struct ad_scan_desc** werden die globalen Parameter wie Abtastzeit und Anzahl der Messwerte festgelegt. Für jeden abzutastenden Kanal ist einmal **struct ad_scan_cha_desc** auszufüllen, darin werden die kanalspezifischen Daten wie Kanalnummer oder Triggereinstellungen definiert.

5.2.1 struct ad_scan_cha_desc

Die Struktur **struct ad_scan_cha_desc** hat folgenden Aufbau:

C

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

Die Elemente der Struktur haben folgende Bedeutung:

cha

Legt die Nummer des Kanals fest, der abgetastet und gespeichert werden soll. Diese ist Hardware abhängig und in den entsprechenden Abschnitten des Kapitels "Messsysteme" (siehe S. 46) beschrieben.

range

© BMC Messsysteme GmbH

Legt den Messbereich des Kanals fest. Die Nummer des Messbereichs ist Hardware abhängig und in den entsprechenden Abschnitten des Kapitels "Messsysteme" (siehe S. 46) beschrieben.

store

Legt zusammen mit **ratio** (siehe u.) fest, wie der Kanal gespeichert wird. Eine ausführliche Beschreibung der Speicherarten folgt im nächsten Abschnitt (siehe "Speicherarten", S. 23).

ratio

Legt das Speicherintervall fest (siehe "Speicherarten", S. 23).

zero

Legt den Nullpegel für die RMS Berechnung fest und wird deswegen auch nur benötigt, wenn der Effektivwert des Signals gespeichert wird.

trg_mode

Legt zusammen mit **trg_par[]** (siehe u.) fest, ob und wie dieser Kanal einen Trigger auslösen soll.

trg_par[]

Legt die Triggerschwellen fest.

samples_per_run

Wird von LIBAD4 zurückgegeben und liefert die Anzahl der Messwerte, die für diesen Kanal produziert werden.

Nicht verwendete bzw. undokumentierte Elemente der Struktur müssen unbedingt auf 0 gesetzt werden!

5.2.1.1 Speicherarten

Kanäle können unterschiedlich gespeichert werden. Die Speicherart wird durch **ratio** und **store** aus der Struktur **struct ad_scan_cha_desc** festgelegt.

Im einfachsten Fall steht **store** auf **AD_STORE_DISCRETE** und **ratio** auf **1**. Dadurch wird jeder erfasste Messwert im Abtasttakt gespeichert:

Zeit	0ms	2ms	4ms	6ms	8ms	10ms	12ms	14ms	16ms	18ms	20ms	22ms	...
Erfassung	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	...
Speicherung	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	...

Neben dem erfassten Messwert lassen sich auch Mittelwert, Minimum, Maximum oder RMS über ein Intervall speichern. Dazu definiert die LIBAD4 folgende Konstanten:

C

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

Folgende Tabelle veranschaulicht den Zusammenhang zwischen dem Abtasttakt und **ratio**. In diesem Beispiel ist die Abtastzeit auf 2ms eingestellt und der Mittelwert des Kanals **a** wird im Verhältnis 1:5 gespeichert (d. h. **store** steht auf **AD_STORE_AVERAGE** und **ratio** auf **5**).

© BMC Messsysteme GmbH

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

Es ist auch möglich, mehrere Werte eines Kanals zu speichern. Folgendes Beispiel zeigt die Speicherung des zuletzt erfassten Werts und des Mittelwerts aus 5 Messwerten (dazu wird **ratio** auf **5** und **store** auf **AD_STORE_DISCRETE|AD_STORE_AVERAGE** gesetzt):

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

5.2.1.2 Triggerarten

Die LIBAD4 definiert folgende Trigger:

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
```

Es kann für jeden einzelnen Kanal ein Trigger definiert werden. Die einzelnen Triggerbedingungen werden mit **or** verknüpft, d. h. der erste Kanal, auf dem die Triggerbedingung erfüllt ist, löst den Trigger des Messsystems aus.

Kanäle, die keinen Trigger auslösen sollen, sollten **trg_mode** auf **AD_TRG_NONE** gesetzt haben. Sind alle Kanäle einer Messung auf **AD_TRG_NONE** gesetzt, wird diese ohne Trigger durchgeführt, d. h. die Messwerte werden sofort gespeichert.

Werden alle Kanäle auf **AD_TRG_NEVER** gestellt, dann wird kein Trigger ausgelöst. In diesem Fall läuft die Messung bis zum expliziten Aufruf der Funktion **ad_stop_scan()**.

Bei den Triggerbedingungen **AD_TRG_POSITIVE** (Positive Flanke) bzw. **AD_TRG_NEGATIVE** (Negative Flanke) wird ein Trigger ausgelöst bei Über- bzw. Unterschreiten des unter "Triggerpegel 1" definierten Messwerts (**struct ad_scan_cha_desc**, Parameter **trg_par[0]**). Bei Messsystemen mit 12- und 16-Bit Auflösung müssen die Werte für den 16-Bit Triggerpegel in den unteren 16-Bit des Triggerpegel Parameters eingetragen werden. Bei Flankentriggerung, wie zum Beispiel "Positive Flanke", muss der Kanal vorher unter dem Triggerpegel sein, bevor die Überschreitung des Pegels den Trigger auslöst.

Bei den Triggerbedingungen **AD_TRG_INSIDE** bzw. **AD_TRG_OUTSIDE** wird ein Trigger ausgelöst, wenn der Messwert innerhalb bzw. außerhalb des durch "Triggerpegel 1" (**struct ad_scan_cha_desc**, Parameter **trg_par[0]** für Minimum) und "Triggerpegel 2" (**struct ad_scan_cha_desc**, Parameter **trg_par[1]** für Maximum) definierten Bereichs liegt. Bei einem Fenstertrigger ist im Unterschied zum Flankentrigger nur der aktuelle Messwert ausschlaggebend für das Auslösen des Triggers.

5.2.2 struct ad_scan_desc

Die globalen Einstellungen eines Scanvorgangs werden in der Struktur **struct ad_scan_desc** festgelegt. Diese hat folgenden Aufbau:

C

```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;
  ...
};
```

Die Elemente der Struktur haben folgende Bedeutung:

sample_rate

Legt die Abtastrate der Messung fest (in Sekunden). Um z. B. eine Abtastrate von 100Hz zu erreichen, muss der Wert 0.01 verwendet werden.

prehist

Legt die Länge der Vorgeschichte fest (nur bei Trigger, sonst auf 0 setzen).

posthist

Legt die Länge der Nachgeschichte fest.

ticks_per_run

Wird für kontinuierliche Messungen benötigt und legt dabei die Blockgröße fest, mit der die Messwerte von dem Messsystem abgeholt werden sollen. Anschließend gibt die LIBAD4 in **ticks_per_run** zurück, mit welcher Blockgröße im Messwertbuffer die Werte von dem Gerät geliefert werden.

bytes_per_run

Wird von LIBAD4 zurückgegeben, legt bei nicht aktivierter interner Messdaten-Speicherverwaltung die Größe des Buffers für **ad_get_next_run()** fest (in Bytes).

samples_per_run

Wird von LIBAD4 zurückgegeben, legt bei nicht aktivierter interner Messdaten-Speicherverwaltung die Anzahl der Messwerte eines Buffers fest, der von **ad_get_next_run_f()** zurückgegeben wird.

flags

Das Bit **AD_SF_SAMPLES** in **flags** legt die Messdaten-Speicherverwaltung fest. Wenn das Bit **AD_SF_SAMPLES** gesetzt ist, wird die interne Messdaten-Speicherverwaltung aktiviert. Wenn das Bit **AD_SF_SAMPLES** nicht gesetzt ist, muss eine eigene Speicherverwaltung realisiert werden.

- **Nicht verwendete bzw. undokumentierte Elemente der Struktur müssen unbedingt auf 0 gesetzt werden!**
- **Die interne Messdaten-Speicherverwaltung kann nur für Messsysteme verwendet werden, die selbstständig die Messdaten erfassen und speichern (z. B. LAN-AD16fx / AMS42/84-LAN, USB-AD16f / AMS42/84-USB, PCIe-BASE / PCI-BASEII / PCI-PIO mit MDA16-4i/-8i/MADDA16/16n).**

© BMC Messsysteme GmbH

- **Bei aktivierter interner Messdaten-Speicherverwaltung muss die Routine ad_poll_scan_state() kontinuierlich aufgerufen werden. Das Auslesen der Messwerte aus dem internen Speicher erfolgt mit den Routinen ad_get_samples(), ad_get_samples_f(), oder ad_get_samples_f64().**

5.2.3 struct ad_scan_state

Während einer laufenden Messung liefert die LIBAD4 den Zustand der Messung in der Struktur **struct ad_scan_state** zurück:

C

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

Die Elemente der Struktur haben folgende Bedeutung:

flags

Zeigt den Zustand der Messung an (siehe u.).

posthist

Enthält die aktuelle Anzahl der Messwerte nach dem Trigger. Ist kein Trigger eingestellt, dann wird die Anzahl der aktuell gesampelten Messwerte übergeben.

runs_pending

Zeigt an, ob der nächste RUN ausgelesen werden kann. Ist dieses Flag ungleich null, dann kann der nächste RUN mit **ad_get_next_run ()** ausgelesen werden.

Im Element **flags** wird der Zustand des Scans übergeben. Damit lässt sich abfragen, ob der Trigger bereits erfolgt ist und ob die Messung noch läuft:

C

```c
struct ad_scan_state state;

...

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

...

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

Die Struktur **struct ad_scan_state** kann von der LIBAD4 entweder beim Auslesen der Messwerte mit **ad_get_next_run()** oder durch den expliziten Aufruf von **ad_poll_scan_state()** erfragt werden.

5.2.4 struct ad_scan_pos

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

Die Struktur **struct ad_scan_pos** enthält RUN Informationen über den Scan.

Die Elemente der Struktur haben folgende Bedeutung:

run

Nummer des RUNs im Scan

offset

Offset in dem jeweiligen RUN des Scans

5.2.5 struct ad_cha_layout

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

Die Struktur **struct ad_cha_layout** enthält Informationen für jeden Kanal, der gescannt wird. Diese Informationen sind erst nach dem Beenden eines Scans vorhanden.

Die Elemente der Struktur haben folgende Bedeutung:

start

Position des ersten Messwertes in dem Scan

prehist

Anzahl der Messwerte vor der Triggerung

posthist

Anzahl der Messwerte nach der Triggerung

t0

Zeit bis zum Eintreten des Triggerereignisses in Sekunden

5.3 Memory-only Scan

Ein "memory-only"-Scan wird durch den Aufruf der drei Funktionen **ad_start_mem_scan()**, **ad_get_next_run()** und **ad_stop_scan()** ausgelöst und durchgeführt. Alle gespeicherten Samples eines solchen Scans liegen im (physikalisch vorhandenen) Hauptspeicher des PCs.

Der Beispielcode im folgenden Kapitel demonstriert das Starten eines Scans und das Auslesen der Messwerte.

© BMC Messsysteme GmbH

5.3.1 Starten eines Scans

Um die Funktion **ad_start_mem_scan()** aufrufen zu können, müssen zuerst die abzutastenden Kanäle definiert werden. Folgendes Beispiel legt die Kanalbeschreibung für zwei Kanäle (Analogeingang 1 und Analogeingang 3) an. Beide Kanäle werden 1:1 gespeichert.

C

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

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

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

Außerdem müssen die globalen Scanparameter in der Struktur **struct ad_scan_desc** gesetzt werden. Folgendes Beispiel setzt die Abtastrate auf 1kHz und speichert 500 Messwerte (pro Kanal).

C

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

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

Anschließend kann **ad_start_mem_scan()** aufgerufen werden:

C

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

Jetzt läuft der Scanvorgang im Hintergrund und ist nach 0.5sec. fertig (500x 1ms).

5.3.2 Auslesen der Messwerte

Das Auslesen der Messwerte geschieht mit der Funktion **ad_get_next_run()** oder **ad_get_next_run_f()**. Dabei liefert **ad_get_next_run()** die Messwerte direkt vom Messsystem (also als 16-Bit Werte). Die Funktion **ad_get_next_run_f()** liefert dagegen Float-Werte, die bereits (je nach Messbereich) in die zugehörigen Spannungswerte umgerechnet sind. Beide Funktionen blockieren im Fall eines "memory-only"-Scans solange, bis alle Messwerte erfasst sind (in unserem Fall also für 0.5 Sekunden).

Beide Funktionen erwarten einen Zeiger auf einen Datenbuffer. Dieser muss groß genug sein, um die gesamten Messwerte aufnehmen zu können, andernfalls wird Speicher überschrieben und das Programm wird abstürzen!

Die Mindestgröße des Buffers für **ad_get_next_run()** lässt sich am Element **bytes_per_run** der Struktur **struct ad_scan_desc** feststellen. Ein Buffer, der von **ad_get_next_run_f()** gefüllt werden soll, muss mindestens für **samples_per_run** Float-Werte Platz bieten.

In unserem Fall werden 2 Kanäle à 500 Messwerte gespeichert, so dass der Messwertspeicher eine Größe von 1000 Float-Werten aufweisen muss:

C

```c
float samples[1000];
...
ASSERT (sd.samples_per_run <= 1000);

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

...
```

Das Feld **samples[]** ist nach dem erfolgreichen Aufruf der Funktion mit folgenden Messwerten beschrieben (die Messwerte $\mathbf{a}_i$ kommen vom Analogeingang 1, die Messwerte $\mathbf{b}_i$ von Analogeingang 3):

Feldindex	0	1	2	…	498	499	500	501	502	…	998	999	...
Zeit (in ms)	**0**	**1**	**2**	**…**	**498**	**499**	**0**	**1**	**2**	**…**	**498**	**499**	...
Messwerte	a_1	a_2	a_3	**…**	a_{499}	a_{500}	b_1	b_2	b_3	**…**	b_{499}	b_{500}	...

5.3.3 Stoppen des Scans

Wenn ein Scanvorgang erfolgreich gestartet wurde (Rückgabewert von **ad_start_scan()** war 0), muss dieser Scan mit **ad_stop_scan()** abgeschlossen werden.

Der Scan muss auch dann gestoppt werden, wenn das Auslesen der Messwerte einen Fehler geliefert hat. Solange der Scan nicht gestoppt worden ist, kann kein neuer Scan gestartet werden.

Folgender Beispielcode stoppt den Scan:

 © BMC Messsysteme GmbH

C

```
int32_t scan_result;
...

rc = ad_stop_scan (adh, &scan_result);

...
```

5.4 Kontinuierliche Messung

Neben dem "memory-only"-Scan bietet die LIBAD4 auch die Möglichkeit, eine kontinuierliche Messung zu starten. Diese hat gegenüber dem "memory-only"-Scan die Eigenschaft, dass die Messwerte blockweise an den Aufrufer übergeben werden. Dadurch ist der Aufrufer in der Lage, die Messwerte während des Scans zu untersuchen, um z. B. eine Regelung durchzuführen.

In diesem Fall werden die Messwerte zu so genannten RUNs zusammengefasst und von der LIBAD4 als RUNs an den Aufrufer übergeben. Die Anzahl der Messwerte, die zu einem RUN zusammengefasst werden, lässt sich durch den Aufrufer durch das Element **ticks_per_run** der Struktur **struct ad_scan_desc** vorgeben.

Dieser Parameter kann durchaus extreme Werte annehmen. Wird **ticks_per_run** beispielsweise auf 1 gesetzt, erzeugt die LIBAD4 für jeden Messwert einen einzelnen RUN. Allerdings lassen sich mit dieser Einstellung selbstverständlich nur noch sehr niedrige Abtastraten realisieren.

Es ist die Aufgabe des Aufrufers, die Anzahl der Messwerte pro RUN so einzustellen, dass **ad_get_next_run()** noch oft genug aufgerufen werden kann, um einen Überlauf der Messwerte zu verhindern. Andernfalls wird die Messung von der LIBAD4 abgebrochen.

5.4.1 Aufbau eines RUNs

Die Anzahl der Messwerte eines RUNs wird an die LIBAD4 im Element **ticks_per_run** der Struktur **struct ad_scan_desc** übergeben. Folgendes Beispiel verteilt die Messwerte des Scans auf zwei RUNs à 250 Messwerte (pro Signal).

Wie dies Beispiel zeigt, wird eine kontinuierliche Abtastung mit **ad_start_scan()** (im Gegensatz zu **ad_start_mem_scan()**) gestartet. In diesem Fall muss das Feld **ticks_per_run** der Struktur **struct ad_scan_desc** vorher definiert werden.

Das Beispiel produziert die folgenden zwei RUNs während der Messung, wobei der erste RUN 250ms nach Start des Scans, der zweite 500ms nach Start des Scans von **ad_get_next_run()** zurückgegeben wird.

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);
...
```

Feldindex	0	1	2	…	48	49	50	51	52	…	98	99	…
Zeit (in ms)	**0**	**1**	**2**	…	**248**	**249**	**0**	**1**	**2**	…	**248**	**249**	…
Messwerte	a_1	a_2	a_3	…	a_{249}	a_{250}	b_1	b_2	b_3	…	b_{249}	b_{250}	…

RUN #0

Feldindex	0	1	2	…	248	249	250	251	252	…	498	499	…
Zeit (in ms)	**250**	**251**	**252**	…	**498**	**499**	**250**	**251**	**252**	…	**498**	**499**	…
Messwerte	a_{251}	a_{252}	a_{253}	…	a_{499}	a_{500}	b_{251}	b_{252}	b_{253}	…	b_{499}	b_{500}	…

RUN #1

© BMC Messsysteme GmbH

Folgender Beispielcode liest die RUNs während der Messung aus:

C

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

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

p = data;
state.flags = AD_SF_SCANNING;

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

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

    p += sd.bytes_per_run;
  }

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

5.4.2 Ein Messwert pro RUN

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;

...
```

Wird **ticks_per_run** auf **1** gestellt, dann werden RUNs mit einem Messwert pro Signal erzeugt.

Das obige Beispiel erzeugt 500 RUNs mit folgendem Aufbau:

Feldindex	0	1
Zeit	0	0
Messwert	a_1	b_1

RUN #0

Feldindex	0	1
Zeit	10	10
Messwert	a_2	b_2

RUN #1

Feldindex	0	1
Zeit	4980	4980
Messwert	a_{499}	b_{499}

RUN #498

© BMC Messsysteme GmbH

Feldindex	0	1
Zeit	4990	4990
Messwert	a_{500}	b_{500}

RUN #499

5.4.3 Signale mit unterschiedlicher Speicherrate

Die beiden vorherigen Beispiele (siehe "Ein Messwert pro RUN", S. 33) beschreiben den Aufbau eines RUNs für Signale, die im Verhältnis 1:1 gespeichert werden. Im Folgenden wird ein Beispiel mit der Speicherrate 1:5 erläutert.

Wird ein Signal in einem anderen Verhältnis als 1:1 gespeichert, dann muss zwischen Abtasttakt und Speicherrate unterschieden werden. Die Abtastrate wird für alle Kanäle des Messsystems durch das Element **sample_rate** der Struktur **struct ad_scan_desc** festgelegt. Die Speicherrate kann für jeden Kanal unterschiedlich sein und ergibt sich über den Parameter **ratio** der Strukur **struct ad_scan_desc**, indem die Speicherrate durch **ratio** geteilt wird.

Folgendes Diagramm stellt einen Scan dar, in dem zwei Eingangskanäle mit einer Abtastrate von 2ms (50Hz) abgetastet werden. Der Eingang **a** wird 1:1 gespeichert, der Eingang **b** speichert den Mittelwert über 5 Messwerte.

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

Nachdem in jedem RUN mindestens ein Messwert pro Kanal gespeichert wird, legen die eingestellten Speicherraten die minimale Größe eines RUNs fest.

In diesem Fall besteht der kleinste mögliche RUN aus fünf Abtasttakten (**ticks_per_run** = 5), in dem fünf Messwerte des Eingangskanals **a** enthalten sind, sowie der Mittelwert aus den fünf Messwerten des Eingangs **b**:

Feldindex	0	1	2	3	4
Zeit (in ms)	0	2	4	6	8
Kanal a Kanal b	a_1	a_2	a_3	a_4	a_5 $\frac{1}{5}\sum b_i$

Werden mehrere Abtasttakte zu einem RUN kombiniert, liegen die gespeicherten Werte pro Signal hintereinander (Beispiel für **ticks_per_run** = 250):

Feldindex	0	1	2	...	248	249	250	251	252	...	398	399
Zeit (in ms)	0	2	4	...	496	498	8	18	28	...	488	498
Werte	a_1	a_2	a_3	...	a_{249}	a_{250}	$\frac{1}{5}\sum b_i$	$\frac{1}{5}\sum b_i$	$\frac{1}{5}\sum b_i$	...	$\frac{1}{5}\sum b_i$	$\frac{1}{5}\sum b_i$

5.5 Messung mit Triggerung

Mit der **LIBAD4** kann eine Messung mit Triggerung erfolgen. Dabei muss die interne Messdaten-Speicherverwaltung aktiviert sein (Bit **AD_SF_SAMPLES** des Elements **flags** in der **struct ad_scan_desc** ist gesetzt).

Die Anzahl der Messwerte vor der Triggerung (Vorgeschichte bzw. Prehistory) und die Anzahl der Messwerte nach der Triggerung (Nachgeschichte bzw. Posthistory) können für die Messung in der Struktur **struct ad_scan_desc** frei eingestellt werden.

© BMC Messsysteme GmbH

Ferner lässt sich für jeden Kanal eine Triggerbedingung in der Scankanalstruktur **struct ad_scan_cha_desc** definieren. Trifft eine der Triggerbedingungen zu, erfolgt die Triggerung und die Messung wird nach Ablauf der Nachgeschichte beendet.

Das kontinuierliche Lesen der Messwerte erfolgt mit dem Befehl **ad_poll_scan_state()**. Dabei wird auch der aktuelle Scanzustand abgefragt. Solange das Bit **AD_SF_SCANNING** im Element **flags** der Struktur **struct ad_scan_state** gesetzt ist, läuft die Messung. Sobald das Bit **AD_SF_TRIGGER** gesetzt ist, hat die Messung getriggert, d. h. wenigstens eine der Triggerbedingungen wurde erreicht.

Das Auslesen der Messwerte erfolgt mit den Routinen **ad_get_samples()**, **ad_get_samples_f()**, oder **ad_get_samples_f64()**. Dabei liefert **ad_get_samples()** die Messwerte direkt vom Messsystem**,** **ad_get_samples_f()** oder **ad_get_samples_f64()** liefern dagegen Float-Werte bzw. Double-Werte, die bereits (je nach Messbereich) in die zugehörigen Spannungswerte umgerechnet sind. Bei diesen Routinen können Daten gezielt für einen Scankanal ausgelesen werden. Anzahl und Startposition der auszulesenden Daten werden beim Funktionsaufruf übergeben. Informationen über den Messdatenspeicher für einen Messkanal werden mittels **ad_get_sample_layout()** abgefragt.

Die Installation des Libad4 SDK unter Windows® beinhaltet ein C/C++-Beispiel für die Programmierung eines Scans mit Triggerung.

5.5.1 Parameter der Scankanäle bei Triggerung

Die Triggerbedingung wird für jeden Kanal mit Hilfe der Parameter **trg_mode** und **trg_par** der Struktur **struct ad_scan_cha_desc** definiert.

Das folgende Beispiel definiert einen Trigger mit positiver analoger Flanke:

© BMC Messsysteme GmbH

C

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

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

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

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

#define TRIGGER_VALUE 2.5

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

Das nächste Beispiel definiert einen digitalen Trigger:

C

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

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

Die Scanparameter der Vorgeschichte und Nachgeschichte werden in der Struktur **struct ad_scan_desc** festgelegt, wie im Folgenden beispielhaft gezeigt wird:

C

```
struct ad_scan_desc sd;

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

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

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

Folgender Codeabschnitt zeigt, wie die Daten von dem Messsystem kontinuierlich geholt werden, solange bis die Messung beendet ist. Die Messung wird mittels **ad_start_scan** gestartet.Ein kontinuierlicher Aufruf von **ad_poll_scan_state** ist nötig, damit die Messdaten in den internen Messdatenspeicher geschrieben werden. Die momentanen vorhanden Messwerte können aus dem Messdatenspeicher auch während der Messung extrahiert werde

C

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

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

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

Nach Triggerung der Messung und Ablauf der Nachgeschichte wird die Messung beendet und die Daten von jedem Kanal können aus dem Ringbuffer extrahiert werden.

© BMC Messsysteme GmbH

C

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

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

5.6 Funktionsbeschreibung (Scan)

5.6.1 ad_start_mem_scan

Prototype

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

C

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

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

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

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

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

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

Startet einen "memory-only"-Scan. Der Funktion werden ein Zeiger auf ein Element der Struktur **struct ad_scan_desc** übergeben, die Zahl der abzutastenden Kanäle und ein Feld von Elementen der Struktur **struct ad_scan_cha_desc**.

Aufgrund von Einschränkungen bei den meisten Messkarten müssen die Eingangskanäle unbedingt in aufsteigender Reihenfolge im Feld chav[] angegeben werden! Werden außer Analogeingängen auch Zähler und Digitaleingänge abgetastet, müssen erst alle analogen, dann alle Zähler und schließlich die digitalen Kanäle angegeben werden!

Die Felder **sample_rate**, **ticks_per_run**, **bytes_per_run** und **samples_per_run** der Struktur **struct ad_scan_desc** werden für die angegebenen Parameter neu berechnet (siehe "ad_calc_run_size", S. 43).

© BMC Messsysteme GmbH

5.6.2 ad_start_scan

Prototype

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

Anders als **ad_start_mem_scan()** wertet **ad_start_scan()** das Element **ticks_per_run** der Struktur **struct ad_scan_desc** aus. Damit kann ein Scan auf mehrere RUNs verteilt werden (siehe "Kontinuierliche Messung", S. 30).

Aufgrund von Einschränkungen bei den meisten Messkarten müssen die Eingangskanäle unbedingt in aufsteigender Reihenfolge im Feldchav[] angegeben werden! Werden außer Analogeingängen auch Zähler und Digitaleingänge abgetastet, müssen erst alle analogen, dann alle Zähler und schließlich die digitalen Kanäle angegeben werden!

Die Felder **sample_rate**, **ticks_per_run**, **bytes_per_run** und **samples_per_run** der Struktur **struct ad_scan_desc** werden für die angegebenen Parameter neu berechnet (siehe "ad_calc_run_size", S. 43).

5.6.3 ad_get_sample_layout

Prototype

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

Liefert bei aktivierter interner Messdaten-Speicherverwaltung Informationen über den Messdatenspeicher für den Scankanal **index**. Die Scankanal Nummerierung beginnt mit **index** = 0.

Die Struktur **struct ad_sample_layout** besteht aus:

C

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

Die Elemente der Struktur haben folgende Bedeutung:

buffer_start

Position des ersten Messwerts im Messdatenspeicher des Scans

start

Position des ersten Messwerts der Vorgeschichte im Messdatenspeicher

prehist_samples

Anzahl der vorhandenen Messwerte vor der Triggerung im Messdatenspeicher. Die Vorgeschichte erstreckt sich von der Position **start** bis zu (**start** + **prehist_samples**).

posthist_samples

Anzahl der vorhandenen Messwerte nach der Triggerung. Die Nachgeschichte erstreckt sich von der Position (**start** + **prehist_samples**) bis zu (**start** + **prehist_samples** + **posthist_samples**).

Die interne Messdaten-Speicherverwaltung (Bit AD_SF_SAMPLES des Elements flags in der struct ad_scan_desc gesetzt) muss aktiviert sein, damit diese Routine benutzt werden kann.

5.6.4 ad_get_samples

Prototype

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

Liefert bei aktivierter interner Messdaten-Speicherverwaltung die direkt vom Messystem gelieferten Messwerte für den Scankanal **index**. Die Scankanal Nummerierung beginnt mit **index** = 0.

Je nach Speichertiefe des Messkanals werden vom Messsystem 16-Bit bzw. 32-Bit Messwerte geliefert. Ab der Position **offset** werden **n** Messwerte aus dem Messdatenspeicher des Scankanals **index** gelesen. Die Position **offset** kann nicht kleiner sein als **buffer_start**, das Element der Struktur **struct ad_get_sample_layout**.

Die wirkliche Anzahl gelesener Messwerte wird in dem Parameter **n** zurück geliefert. Der Parameter **type** entscheidet, welche Daten aus dem Datenspeicher in dem zur Verfügung gestellten Bereich **buf** geschrieben werden. Es können nur die Datentypen (Discrete, Minimum, Maximum, etc) extrahiert werden, die bei dem Speichermodus (Element **store** in **struct ad_scan_cha_desc**) des Scankanals definiert waren.

Die Funktion erwartet einen Zeiger auf einen Datenbuffer. Dieser muss groß genug sein, um die gesamten Messwerte aufnehmen zu können. Andernfalls wird Speicher überschrieben und das Programm wird abstürzen!

Die interne Messdaten-Speicherverwaltung (Bit AD_SF_SAMPLES des Elements flags in der struct ad_scan_desc gesetzt) muss aktiviert sein, damit diese Routine benutzt werden kann.

5.6.5 ad_get_samples_f

Prototype

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

Liefert bei aktivierter interner Messdaten-Speicherverwaltung die Messwerte als Float- bzw. Double-Werte für den Scankanal **index**. Die Scankanal Nummerierung beginnt mit **index** = 0.

© BMC Messsysteme GmbH

Die vom Messsystem gelieferten Messwerte wurden je nach eingestelltem Messbereich in die zugehörigen Spannungswerte umgerechnet. Ab der Position **offset** werden **n** Messwerte im Float-Format aus dem Messdatenspeicher des Scankanals **index** gelesen. Die Position **offset** kann nicht kleiner sein als **buffer_start**, das Element der Struktur **struct ad_get_sample_layout**.

Die wirkliche Anzahl gelesener Messwerte wird in dem Parameter **n** zurück geliefert. Der Parameter **type** entscheidet, welche Daten aus dem Datenspeicher in dem zur Verfügung gestellten Bereich **buf** geschrieben werden. Es können nur die Datentypen (Discrete, Minimum, Maximum, etc) extrahiert werden, die bei dem Speichermodus (Element **store** in **struct ad_scan_cha_desc**) des Scankanals definiert waren.

- **Die Funktion erwartet einen Zeiger auf einen Datenbuffer. Dieser muss groß genug sein, um die gesamten Messwerte aufnehmen zu können. Andernfalls wird Speicher überschrieben und das Programm wird abstürzen!**
- **Die interne Messdaten-Speicherverwaltung (Bit AD_SF_SAMPLES des Elements flags in der struct ad_scan_desc gesetzt) muss aktiviert sein, damit diese Routine benutzt werden kann.**
- **Werden Messkanäle mit einer Speichertiefe von mehr als 16 Bit verwendet (z. B. 32-Bit Zähler der USB-OI16 / PCIe-BASE / PCI-BASEII / PCI-PIO), sollte die Routine ad_get_samples_f64() benutzt werden.**

5.6.6 ad_get_samples_f64

Prototype

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

Liefert bei aktivierter interner Messdaten-Speicherverwaltung die Messwerte als Float- bzw. Double-Werte für den Scankanal **index**. Die Scankanal Nummerierung beginnt mit **index** = 0.

Die vom Messsystem gelieferten Messwerte wurden je nach eingestelltem Messbereich in die zugehörigen Spannungswerte umgerechnet. Ab der Position **offset** werden **n** Messwerte im Float-Format aus dem Messdatenspeicher des Scankanals **index** gelesen. Die Position **offset** kann nicht kleiner sein als **buffer_start**, das Element der Struktur **struct ad_get_sample_layout**.

Die wirkliche Anzahl gelesener Messwerte wird in dem Parameter **n** zurück geliefert. Der Parameter **type** entscheidet, welche Daten aus dem Datenspeicher in dem zur Verfügung gestellten Bereich **buf** geschrieben werden. Es können nur die Datentypen (Discrete, Minimum, Maximum, etc) extrahiert werden, die bei dem Speichermodus (Element **store** in **struct ad_scan_cha_desc**) des Scankanals definiert waren.

- **Die Funktion erwartet einen Zeiger auf einen Datenbuffer. Dieser muss groß genug sein, um die gesamten Messwerte aufnehmen zu können. Andernfalls wird Speicher überschrieben und das Programm wird abstürzen!**
- **Die interne Messdaten-Speicherverwaltung (Bit AD_SF_SAMPLES des Elements flags in der struct ad_scan_desc gesetzt) muss aktiviert sein, damit diese Routine benutzt werden kann.**

© BMC Messsysteme GmbH

5.6.7 ad_calc_run_size

Prototype

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

Berechnet die Felder **sample_rate**, **ticks_per_run**, **bytes_per_run** und **samples_per_run** der Struktur **struct ad_scan_desc** für die angegebenen Parameter.

Die Felder werden wie bei einem Aufruf der Funktion **ad_start_scan()** berechnet, allerdings ohne den Scanvorgang auszulösen. Wie durch **ad_start_scan()** erfolgt die Berechnung bzw. Anpassung folgendermaßen:

sample_rate

Wird auf die tatsächlich mögliche Abtastzeit gestellt (die meisten Messkarten können die Abtastzeit nur in festen Schritten einstellen).

ticks_per_run

Wird so angepasst, dass mindestens ein Wert jedes Signals gespeichert wird und/oder ein einzelner RUN in den internen Speicher des Treibers passt.

bytes_per_run

Wird von LIBAD4 berechnet und gibt für **ad_get_next_run()** die Anzahl der Bytes des Buffers vor.

samples_per_run

Wird von LIBAD4 berechnet und gibt für **ad_get_next_run_f()** die Anzahl der Floatwerte innerhalb eines Buffers vor.

Aus **samples_per_run** lässt sich die Größe des Buffers für **ad_get_next_run_f()** berechnen:

C

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

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

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

5.6.8 ad_get_next_run

Prototype

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


© BMC Messsysteme GmbH

Liefert die Messwerte eines Scans.

Die Funktion **ad_get_next_run()** liefert die Messwerte direkt vom Messsystem (also als 16-Bit oder 32-Bit Werte, je nach Speichertiefe des Messkanals). Die untere Messbereichsgrenze entspricht dem Wert **0x0000**, die obere Messbereichsgrenze dem Wert **0xffff** bzw. **0xffffffff** (genauer gesagt entspricht die obere Grenze dem Wert **0x10000** bzw. **0x100000000**, der nicht erreicht wird).

Die Messwerte werden in "network byte order" geliefert, sind also nicht in der byte order einer x86 CPU!

Die Funktion blockiert solange, bis die Messwerte eines RUNs eingetroffen sind. Dies bedeutet, dass die Funktion bei einem "memory-only"-Scan bis zum Ende der Messung blockiert (da ein "memory-only"-Scan alle Messwerte in einem RUN speichert).

5.6.9 ad_get_next_run_f

Prototype

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

Liefert die Messwerte eines Scans als Float-Werte.

Die vom Messsystem gelieferten Messwerte werden je nach Messbereich in die zugehörigen Spannungswerte umgerechnet.

Die Funktion blockiert solange, bis die Messwerte eines RUNs eingetroffen sind. Dies bedeutet, dass die Funktion bei einem "memory-only"-Scan bis zum Ende der Messung blockiert (da ein "memory-only"-Scan alle Messwerte in einem RUN speichert).

Werden Messkanäle mit einer Speichertiefe von mehr als 16 Bit verwendet (z. B. 32-Bit Zähler der USB-OI16 /PCIe-BASE / PCI-BASEII / PCI-PIO), sollte die Routine ad_get_next_run_f64() benutzt werden.

5.6.10 ad_get_next_run_f64

Prototype

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

Liefert die Messwerte eines Scans als Float-Werte.

Die vom Messsystem gelieferten Messwerte werden je nach Messbereich in die zugehörigen Spannungswerte umgerechnet.

Die Funktion blockiert solange, bis die Messwerte eines RUNs eingetroffen sind. Dies bedeutet, dass die Funktion bei einem "memory-only"-Scan bis zum Ende der Messung blockiert (da ein "memory-only"-Scan alle Messwerte in einem RUN speichert).

5.6.11 ad_poll_scan_state

Prototype

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

Liefert den aktuellen Zustand der Messung wie ein Aufruf der Funktion **ad_get_next_run()**. Im Gegensatz zu **ad_get_next_run()** blockiert die Funktion nicht.

Bei aktivierter interner Messdaten-Speicherverwaltung (Bit AD_SF_SAMPLES des Elements flags in der struct ad_scan_desc gesetzt) muss die Routine ad_poll_scan_state() kontinuierlich aufgerufen werden.

5.6.12 ad_stop_scan

Prototype

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

Beendet den Scan. In **scan_result** wird das Ergebnis des Scans übergeben (z. B. eine Fehlernummer, wenn der Scan wegen Überlauf abgebrochen wurde).

Wenn ein Scanvorgang erfolgreich gestartet wurde (Rückgabewert von ad_start_scan() war 0), muss dieser Scan mit ad_stop_scan () abgeschlossen werden.

 © BMC Messsysteme GmbH

6 Messsysteme

6.1 Hinweise

Ein- bzw. Ausgangskanäle werden in LIBAD4 durch Kanalnummern spezifiziert. Die Kanalnummer (32-Bit Integer) legt auch die Kanalart fest. Die Kanalart unterscheidet zwischen Analogeingang, Analogausgang und Digitalkanal. Diese Codierung ist im obersten Byte der Kanalnummer vorhanden und muss per "oder"-Operator (|) mit der Kanalnummer verknüpft werden.

Folgende Kanalarten sind in LIBAD4 definiert:

C

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

Die Kanalnummern sind abhängig vom eingesetzten Messsystem und in den entsprechenden Kapiteln dokumentiert. Der erste Analogeingang eines USB-AD14f lässt sich z. B. angeben durch den Ausdruck **AD_CHA_TYPE_ANALOG_IN|1**.

Analoge Kanäle erwarten neben der Kanalnummer noch die Angabe eines Messbereichs (bzw. Ausgangsbereichs), in dem gemessen (bzw. ausgegeben) werden soll. Dieser Messbereich ist wie die Kanalnummer vom Messsystem abhängig und in den folgenden Kapiteln dokumentiert.

6.2 LAN-AD16fx / AMS42/84-LAN16fx

Um ein LAN-Messsystem vom Typ LAN-AD16fx und AMS42/84-LAN16fx mit der LIBAD4 zu öffnen, wird an ad_open**()** der String "**lanbase:<ip-addr>**" oder "**lanbase:@<sn>**" übergeben. Dabei muss **<ip-addr>** durch die entsprechende IP-Adresse ersetzt werden oder **<sn>** durch die Seriennummer des LAN-AD16f(x). Der String "**lanbase:192.168.1.1**" öffnet z. B. das LAN-Gerät mit der IP Adresse 192.168.1.1 und der String "**lanbase:@157**" öffnet das LAN-Gerät mit der Seriennummer 157.

Mess-system	Analog	Kanal-nummer	range (Messber.)	range (Aus-gabebereich)	Digital	Rich-tung
LAN-AD16fx/ AMS42/84-LAN16fx	16 Eingänge 2 Ausgänge	1..16 1 .. 2	0 (±1.024V) 1 (±2.048V) 2 (±5.120V) 3 (±10.240V)	0 (±10.24V)	2 Ports (je 16 Bit)	1: Port A 2: Port B

6.2.1 Kanalnummern LAN-AD16fx / AMS42/84-LAN16fx

Die 16 Analogeingänge eines LAN-AD16fx oder AMS42/84-LAN16fx besitzen die Kanalnummern 1-16. Die beiden Analogausgänge haben die Kanalnummern 1 und 2. Damit ergeben sich für die 16 Analogeingänge folgende Konstanten:

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)
```

© BMC Messsysteme GmbH

Die beiden analogen Ausgangskanäle eines LAN-AD16fx und AMS42/84-LAN16fx erhalten die folgenden Konstanten:

C

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

Das LAN-AD16fx oder AMS42/84-LAN16fx stellt zwei 16-Bit Digitalports zur Verfügung. Die Richtung der digitalen Portleitungen ist in 8er Gruppen umschaltbar (siehe "ad_set_line_direction", S. 20). Nach dem Einschalten steht der erste Port auf Eingang, der zweite auf Ausgang. Es ergeben sich folgende Konstanten:

C

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

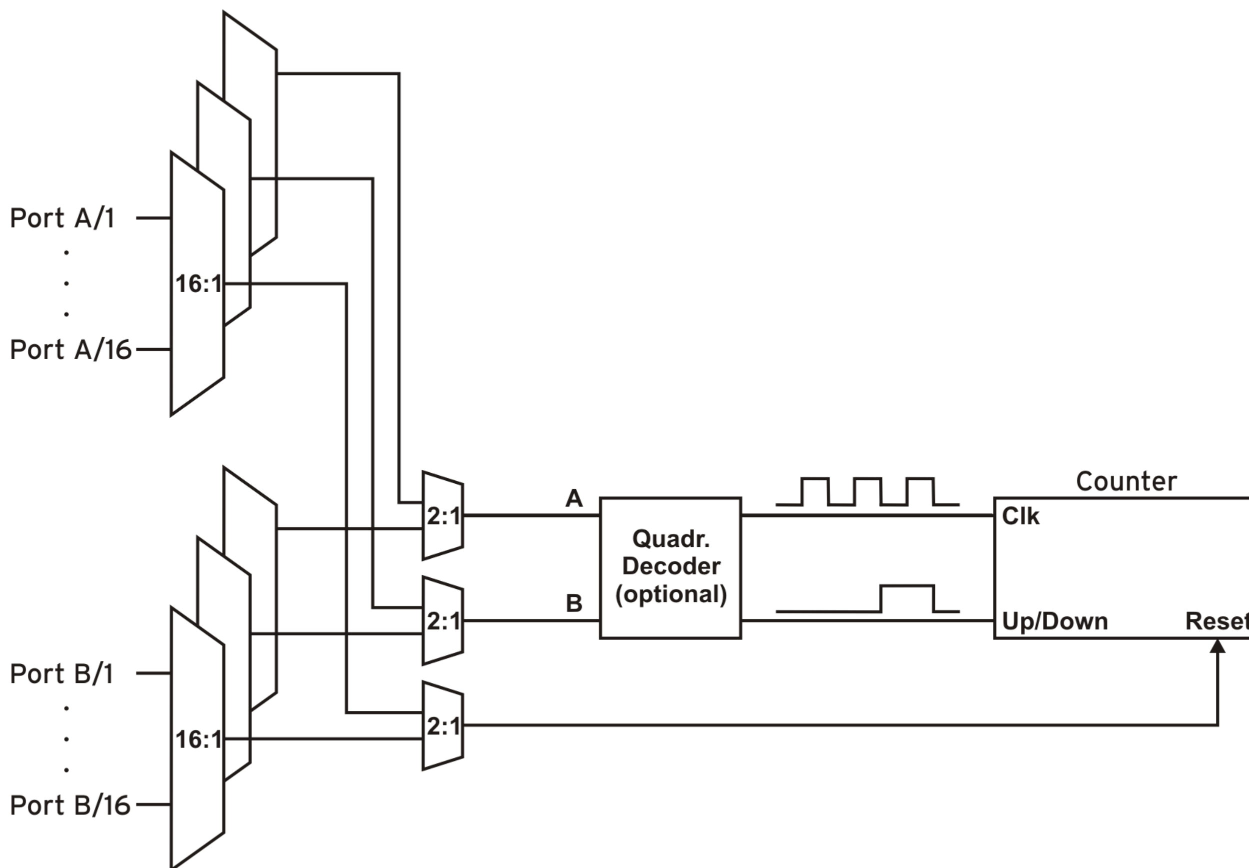
Außerdem stellt das LAN-AD16fx oder AMS42/84-LAN16fx drei 32-Bit Zählereingänge zur Verfügung. Diese können in verschiedenen Betriebsarten verwendet werden und müssen vor der Verwendung per Software konfiguriert werden (siehe "Konfiguration der LAN-AD16fx / AMS42/84-LAN16fx Zähler", S. 48). Die Eingänge der Zähler (Signal A, Signal B, Reset) werden dazu mit beliebigen Digitaleingangsleitungen des LAN-AD16fx oder AMS42/84-LAN16fx verbunden. Für die 32-Bit Zählereingänge ergeben sich folgende Konstanten:

C

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


© BMC Messsysteme GmbH

6.2.2 Konfiguration der LAN-AD16fx / AMS42/84-LAN16fx Zähler



Zur Einstellung der Zähler werden die Konfigurationsparameter in die Struktur **ad_counter_mode** eingetragen und an **ad_ioctl()** übergeben.

Das folgende Beispiel zeigt das prinzipielle Vorgehen: Es konfiguriert den ersten Zähler des LAN-AD16fx und AMS42/84-LAN16fx in der Betriebsart "Zähler" und verbindet den Zählereingang A mit dem zweiten Eingangspin des ersten Digitalports.

Prototype

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

C

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

struct ad_counter_mode par;
int32_t adh;
int32_t st;

...

  adh = ad_open ("pcibase");

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

Die Struktur **ad_counter_mode** hat folgenden Aufbau:

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;
  ...
};
```

Die Elemente der Struktur haben folgende Bedeutung:

cha

Legt den Zählerkanal fest, der konfiguriert werden soll.

mode

Legt die Betriebsart des Zählers fest.

Betriebsart	Beschreibung
AD_CNT_COUNTER	Der Zählerkanal arbeitet als einfacher Zähler. Es wird nur der Eingang A des Zählers verwendet, wobei jede positive Flanke am Eingang den Zählerstand um eins erhöht.
AD_CNT_UPDOWN	Der Zählerkanal arbeitet als Up/Down Zähler, also mit umschaltbarer Zählrichtung. Dabei arbeitet der Eingang A des Zählers als Takteingang, der Eingang B übernimmt die Richtungsumschaltung. Steht der Eingang B des Zählers auf low, dann erhöht jede positive Flanke am Eingang A den Zählerstand um eins. Ansonsten erniedrigt die positive Flanke den Zählerstand.

© BMC Messsysteme GmbH

AD_CNT_QUAD_DECODER	Der Zähler decodiert an den Eingängen A und B die zwei Spuren eines Inkrementalgebers. Dabei wird jede Flanke der beiden Spuren dekodiert.
AD_CNT_PULSE_TIME	Konfiguriert den Zähler für die Pulsweitenmessung. Dabei ist der Zählereingang mit einer internen Taktquelle (60 MHz) verbunden und wird mit jeder Flanke am Eingang A gestartet und gestoppt.

mux_a, mux_b, mux_rst

Legt die Pins der beiden Digitalports fest, die mit den jeweiligen Eingängen des Zählers verbunden sind. Dabei ist es nicht möglich, die Eingänge eines Zählers mit verschiedenen Digitalports zu verbinden (d. h. die Eingänge A, B und *Reset* müssen entweder alle mit Pins aus dem Port A verbunden sein oder alle mit Pins aus dem Port B).

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

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

flags

Legt die Arbeitsweise der Zählereingänge fest. Die Arbeitsweisen können mit **OR** verknüpft werden, z. B. **AD_CNT_INV_RST|AD_CNT_ENABLE_RST**.

Arbeitsweise	Beschreibung
AD_CNT_INV_A	Zählereingang A reagiert invertiert.
AD_CNT_INV_B	Zählereingang A reagiert invertiert.
AD_CNT_INV_RST	Reseteingang reagiert invertiert.
AD_CNT_ENABLE_RST	Reseteingang ist aktiviert.

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

Um eine PCIe-BASE, PCI-BASEII oder PCI-PIO mit der LIBAD4 zu öffnen, muss an ad_open**()** der String "**pcibase**" übergeben werden. Beim Öffnen des Treibers wird nicht zwischen verschiedenen Versionen der PCI(e)-Messkarte unterschieden.

Mehrere Karten lassen sich durch Angabe der Kartennummer unterscheiden (1. Karte mit "**pcibase:0**", 2. Karte mit "**pcibase:1**", usw.).

Eine Messkarte kann auch direkt unter Angabe ihrer Seriennummer geöffnet werden. Die Karte mit der Seriennummer 157 lässt sich zum Beispiel mit "**pcibase:@157**" ansprechen.

6.3.1 Digitalports und Zähler

Die PCIe-BASE / PCI-BASEII / PCI-PIO stellen zwei 16-Bit Digitalports zur Verfügung.

Die Zähler auf der PCIe-BASE / PCI-BASEII / PCI-PIO können in verschiedenen Betriebsarten verwendet werden und müssen vor der Verwendung per Software konfiguriert werden.

Jeder Eingang des Zählers kann beliebig mit einer der je 16 digitalen Leitungen der beiden Digitalports verbunden werden. Auch diese Einstellung muss vor Verwenden des Zählers konfiguriert werden (siehe "Konfiguration der Zähler", S. 52).

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

Die Digitalports der PCIe-BASE / PCI-BASEII / PCI-PIO sind in 8er Gruppen in ihrer Richtung umschaltbar (siehe "ad_set_line_direction", S. 20). Nach dem Einschalten steht der erste Port auf Eingang, der zweite auf Ausgang. Es wird folgende Nummerierung verwendet:

C

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

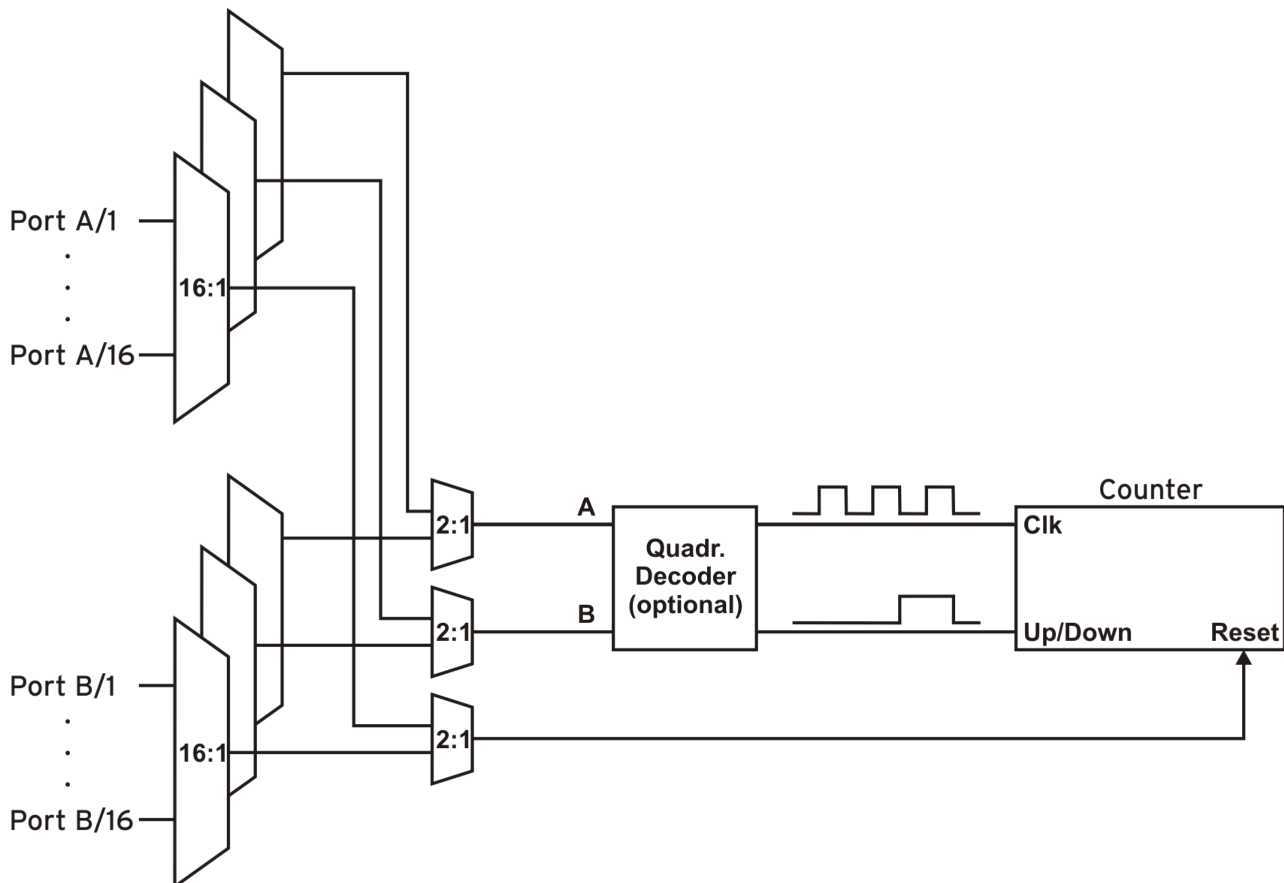
Außerdem stellt die PCIe-BASE / PCI-BASEII / PCI-PIO drei 32-Bit Zählereingänge zur Verfügung:

C

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

 © BMC Messsysteme GmbH

6.3.1.2 Konfiguration der Zähler



Zur Einstellung der Zähler werden die Konfigurationsparameter in die Struktur **ad_counter_mode** eingetragen und an **ad_ioctl()** übergeben.

Das folgende Beispiel zeigt das prinzipielle Vorgehen: Es konfiguriert den ersten Zähler der PCIe-BASE / PCI-BASEII / PCI-PIO in der Betriebsart "Zähler" und verbindet den Zählereingang A mit dem zweiten Eingangspin des ersten Digitalports.

Prototype

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

C

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

struct ad_counter_mode par;
int32_t adh;
int32_t st;

...

  adh = ad_open ("pcibase");

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

Die Struktur **ad_counter_mode** hat folgenden Aufbau:

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;
  ...
};
```

Die Elemente der Struktur haben folgende Bedeutung:

cha

Legt den Zählerkanal fest, der konfiguriert werden soll.

mode

Legt die Betriebsart des Zählers fest.

Betriebsart	Beschreibung
AD_CNT_COUNTER	Der Zählerkanal arbeitet als einfacher Zähler. Es wird nur der Eingang A des Zählers verwendet, wobei jede positive Flanke am Eingang den Zählerstand um eins erhöht.
AD_CNT_UPDOWN	Der Zählerkanal arbeitet als Up/Down Zähler, also mit umschaltbarer Zählrichtung. Dabei arbeitet der Eingang A des Zählers als Takteingang, der Eingang B übernimmt die Richtungsumschaltung. Steht der Eingang B des Zählers auf low, dann erhöht jede positive Flanke am Eingang A den Zählerstand um eins. Ansonsten erniedrigt die positive Flanke den Zählerstand.

© BMC Messsysteme GmbH

AD_CNT_QUAD_DECODER	Der Zähler decodiert an den Eingängen A und B die zwei Spuren eines Inkrementalgebers. Dabei wird jede Flanke der beiden Spuren dekodiert.

mux_a, mux_b, mux_rst

Legt die Pins der beiden Digitalports fest, die mit den jeweiligen Eingängen des Zählers verbunden sind. Dabei ist es nicht möglich, die Eingänge eines Zählers mit verschiedenen Digitalports zu verbinden (d. h. die Eingänge A, B und *Reset* müssen entweder alle mit Pins aus dem Port A verbunden sein oder alle mit Pins aus dem Port B).

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

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

flags

Legt die Arbeitsweise der Zählereingänge fest. Die Arbeitsweisen können mit **OR** verknüpft werden, z. B.

AD_CNT_INV_RST|AD_CNT_ENABLE_RST.

Arbeitsweise	**Beschreibung**
AD_CNT_INV_A	Zählereingang A reagiert invertiert.
AD_CNT_INV_B	Zählereingang A reagiert invertiert.
AD_CNT_INV_RST	Reseteingang reagiert invertiert.
AD_CNT_ENABLE_RST	Reseteingang ist aktiviert.

6.3.2 Aufsteckmodule

Auf der PCIe-BASE / PCI-BASEII / PCI-PIO lassen sich bis zu zwei Aufsteckmodule installieren. Diese Module stellen weitere Kanäle zur Verfügung und sind in den folgenden Kapiteln beschrieben.

6.3.2.1 MADDA16/16n

Der erste analoge Eingangs- oder Ausgangskanal eines MADDA16/16n beginnt bei 1. Befindet sich ein zweites Analogmodul auf der PCI(e)-Multifunktionskarte (nicht: PCI-PIO), wird der erste Eingang bzw. Ausgang des zweiten Moduls unter der Nummer 257 (0x100+1) angesprochen.

Der Steckplatz des Moduls auf der Messkarte ist unerheblich. Lediglich die Moduladresse entscheidet über die Zuordnung der Kanäle. Beispielsweise werden einem MADDA Modul mit niedrigerer Adresse die Kanäle 1 bis 16 (Analogeingänge, Kanalnummern 0x001 bis 0x010) bzw. 1 bis 2 (Analogausgänge, Kanalnummern 0x001 bis 0x002) zugewiesen, dem MADDA Modul mit der höheren Adresse die Kanäle 17-32 (Analogeingänge, Kanalnummern 0x101 bis 0x110) bzw. 3 bis 4 (Analogausgänge, Kanalnummern 0x003 bis 0x004).

Modul	Analog	Kanalnummer	range (Messbereich)	range (Ausgabebereich)
MADDA16, MADDA16n	16 Eingänge 2 Ausgänge	1..16 1..2	0 (±1.024V) 1 (±2.048V) 2 (±5.120V) 3 (±10.240V)	0 (±10.24V)

Damit ergeben sich für die ersten 32 Analogeingänge folgende Konstanten:

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)
```

Befinden sich zwei MADDA-Module auf der Messkarte erhalten die zwei analogen Ausgangskanäle pro MADDA-Modul die folgenden Konstanten:

C

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

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

6.3.2.2 MDA16-4i/-8i

Die Kanäle eines zweiten analogen Ausgangsmoduls werden ab der Nummer 257 (0x100+1) angesprochen. Die Modulreihenfolge wird nur durch die Moduladresse festgelegt und nicht durch den Steckplatz auf der Trägerkarte, d. h. die Kanäle des Moduls mit der höheren Adresse beginnen ab 0x101.

© BMC Messsysteme GmbH

Modul	Analog	Kanalnummer	Ausgabebereich	range
MDA16-4i	4 Ausgänge	1..4	±10.24V	0
MDA16-8i	8 Ausgänge	1..8	±10.24V	0

Die Definition der Kanalnummern ist abhängig von der vorhandenen Kombination der Ausgabemodule auf einer Messkarte. Beispielsweise ergibt sich für ein MDA16-2i und einem MDA16-4i Ausgabemodul auf einer Messkarte die folgende Kanalnummernzuordnung:

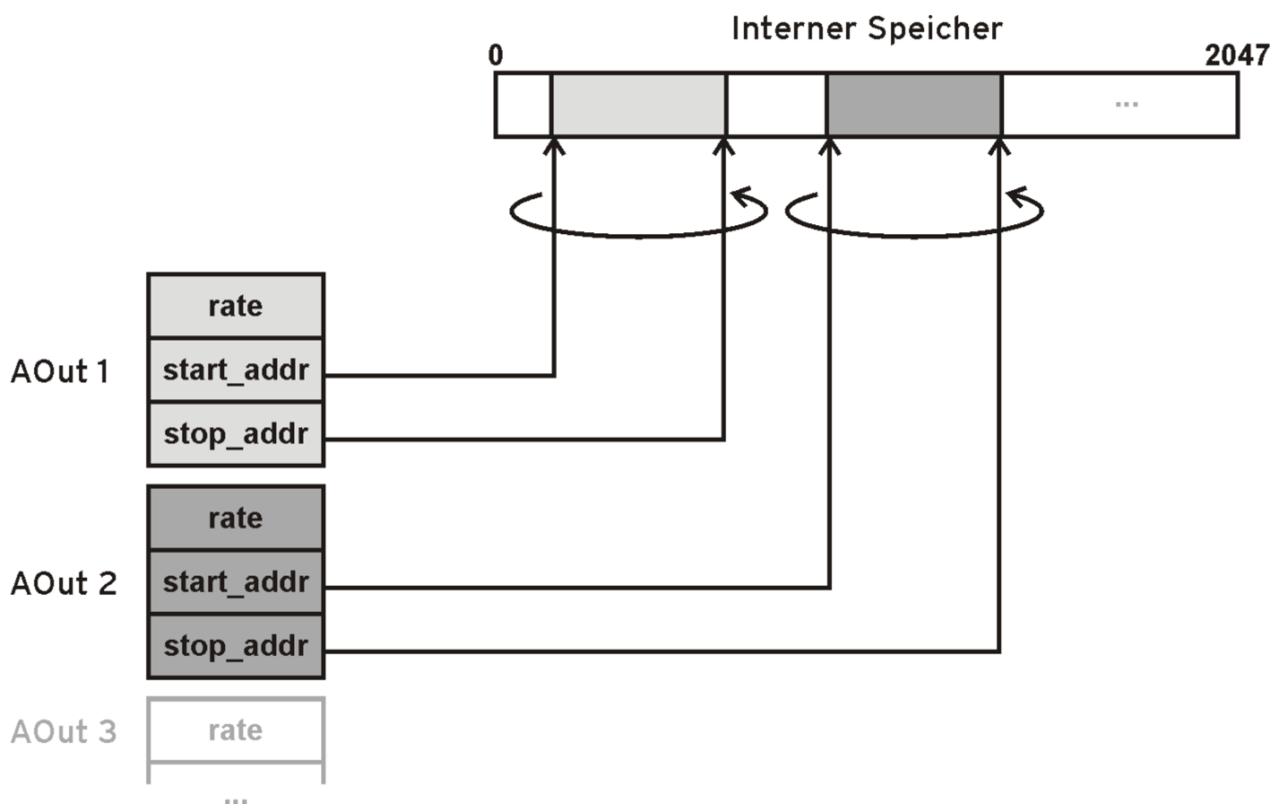
C

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

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

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

6.3.2.3 Funktionsgenerator der MDA16-4i/-8i



Die Ausgabemodule MDA16-4i/-8i enthalten einen Funktionsgenerator, mit dem periodische Signalformen ausgegeben werden können.

Dazu existiert auf dem Modul ein Speicher für 2048 Messwerte, in den beliebige Signalformen geladen werden können. Jeder Ausgabekanal der MDA16-4i/-8i Module besitzt einen eigenen Controller, der einen beliebigen Speicherbereich des internen Speichers ausgeben kann. Sowohl der Speicherbereich als auch die Ausgabefrequenz lässt sich dabei für jeden Kanal getrennt einstellen.

Zur Konfiguration der einzelnen Kanäle werden die Parameter in die Struktur **ad_mda2_generator** eingetragen und mit Hilfe von **ad_ioctl()** übergeben.

Die Struktur **ad_mda2_generator** erlaubt die Definition der Parameter für alle Ausgabekanäle eines Moduls und hat folgenden Aufbau:

C

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

Die Elemente der Struktur haben folgende Bedeutung:

cha

Ausgabekanal des Moduls: Legt fest auf welchem Modul die Ausgabeparameter verändert werden sollen, z. B. für das erste Modul auf einer Messkarte (**AD_CHA_TYPE_ANALOG_OUT|0x0001**) und für ein eventuell vorhandenes zweites Modul (**AD_CHA_TYPE_ANALOG_OUT|0x0101**).

chac

Anzahl der zu definierenden Ausgabecontroller

chav

Ausgabeparameterstrukturen der zu definierenden Ausgabecontroller

ram

Speicher mit Ausgabewerten für die Analogausgabe: Die Analogausgabe ist linear skaliert, wobei der Wert **0x00000000** eines Analogausgangs der niedrigsten Ausgangsspannung, der Wert **0xffffffff** der höchsten Ausgangsspannung entspricht. Mittels ad_float_to_sample**()** lässt sich ein Spannungswert (float) in einen Ausgabewert umrechnen.

Die Struktur **ad_mda2_generator_cha** erlaubt die Definition der Parameter für einen Ausgabekanal und hat folgenden Aufbau:

C

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

Die Speicherdaten von der Startadresse **start_addr** bis zur Stopadresse **stop_addr** werden periodisch vom Ausgabecontroller **cha** unter Berücksichtigung der eingestellten Ausgaberate **rate** auf dem Ausgabekanal ausgegeben. Die jeweiligen Start- und Stopadressen der Ausgabecontroller dürfen sich nicht überlappen.

Die Elemente der Struktur haben folgende Bedeutung:

cha

Controllernummer des Ausgabekanals: Jedes Modul besitzt einen Controller pro Analogausgang. Die Controllernummer wird von 0 ab indiziert (z. B. für einem MDA16-4i mit 4 Ausgabekanälen stehen die Controllernummern 0 bis 3 zur Verfügung). Die Controllernummer 0 entspricht Analogausgang 1, usw.

© BMC Messsysteme GmbH

range

Messbereichsnummer der Ausgabe: Bei einem MDA16-4i/8i mit ±10.24V Messbereich ist die Messbereichsnummer 0.

rate

Teiler für die Ausgabefrequenz: Der Ausgabecontroller wird mit einer Ausgabefrequenz betrieben, die sich aus der maximalen Ausgabefrequenz des Moduls geteilt durch **rate** ergibt. Bei einem MDA16-4i/8i beträgt die maximale Ausgabefrequenz 100kHz. Eine **rate** von 100 wird zu einer Ausgabeauflösung von 1kHz bzw. 1ms pro Ausgabepunkt führen.

start_addr

Startadresse im Speicherbereich des Moduls

stop_addr

Stopadresse im Speicherbereich des Moduls

Das folgende Beispiel zeigt das prinzipielle Vorgehen:

Prototype

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

C

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

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

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

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

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

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

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

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

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

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

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


© BMC Messsysteme GmbH

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

Um ein USB-AD oder eine USB-PIO/USB-PIO-OEM mit der LIBAD4 zu öffnen, muss an ad_open**()** der String "**usb-ad**" bzw. "**usb-pio**" übergeben werden. Mehrere USB Messsysteme lassen sich durch Angabe der Gerätenummer öffnen (1. Gerät mit "**usb-ad:0**", 2. Gerät mit "**usb-ad:1**", usw., bzw. 1. Gerät mit "**usb-pio:0**", 2. Gerät mit "**usb-pio:1**", usw.). Die Reihenfolge der Geräte wird durch das Anstecken bestimmt.

Da USB Messsysteme im Betrieb an- und abgesteckt werden können, ist es möglich, dass die Gerätenummern nicht aufeinander folgend vergeben sind. Steckt man z. B. drei USB-AD an und entfernt dann das 2. Gerät, sind die verbleibenden USB-AD mit "**usb-ad:0**" und "**usb-ad:2**" anzusprechen.

Um unabhängig von dieser Ansteckreihenfolge zu sein, kann ein Gerät auch mit einer bestimmten Seriennummer geöffnet werden. Das Gerät mit der Seriennummer 157 lässt sich zum Beispiel mit "**usb-ad:@157**" bzw. "**usb-pio:@157**" ansprechen.

6.4.1 Eckdaten und Kanalnummern USB-AD / USB-AD-OEM

Mess-system	Analog	Kanal-nummer	range (Messber.)	range (Aus-gabebereich)	Digital	Richtung
USB-AD	16 Eingänge 1 Ausgang	1..16 1	0 (±5.12V)	0 (±5.12V)	2 Ports (je 4 Bit)	1: Eingang (Bit 0..3) 2: Ausg. (Bit 0..3)

Der erste analoge Eingangskanal eines USB-AD / USB-AD-OEM beginnt bei 1. Damit ergeben sich für die 16 Analogeingänge folgende Konstanten:

C

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

Der analoge Ausgangskanal eines USB-AD /USB-AD-OEM erhält die folgende Konstante:

C

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

Aus Kompatibililtätsgründen lässt sich für die Anlogeingänge auch der Messbereich 33 und für den Analogausgang der Ausgabebereich 1 angeben.

Die Richtung der digitalen Portleitungen des USB-AD ist nicht umschaltbar. Dabei stehen die 4 Leitungen des ersten Ports (DIO1) auf Eingang, die 4 Leitungen des zweiten Ports (DIO2) auf Ausgang.

Das USB-AD-OEM stellt zwei Ports mit je acht Leitungen zur Verfügung. Die Portrichtung ist umschaltbar (jeweils für alle acht Leitungen eines Ports gemeinsam). Der erste Port (DIO1) steht per default auf Eingang, der zweite Port (DIO2) auf Ausgang.

Für die Kanäle ergeben sich folgende Konstanten:

© BMC Messsysteme GmbH

C

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

6.4.2 Eckdaten und Kanalnummern USB-PIO(-OEM)

Messsystem	Digital	Kanalnummer
USB-PIO, USB-PIO-OEM	3 Ports (je 8 Bit)	1..3 (Bit 0..7)

Die Richtung der Leitungen ist für jeden Port getrennt einstellbar. Die Umstellung erfolgt portweise (siehe "ad_set_line_direction", S. 20).

C

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

6.5 USB-AD14f

Um ein USB-AD14f mit der LIBAD4 zu öffnen, muss an ad_open**()** der String "**usbad14f**" übergeben werden. Mehrere USB Messsysteme lassen sich durch Angabe der Gerätenummer öffnen (1. USB-AD14f mit "**usbad14f:0**", 2. USB-AD14f mit "**usbad14f:1**", usw.). Die Reihenfolge der Geräte wird durch das Anstecken bestimmt.

Da USB Messsysteme im Betrieb an- und abgesteckt werden können, ist es möglich, dass die Gerätenummern nicht aufeinander folgend vergeben sind. Steckt man z. B. drei USB-AD14f an und entfernt dann das 2. Gerät, sind die verbleibenden USB-AD14f mit "**usbad14f:0**" und "**usbad14f:2**" anzusprechen.

Um unabhängig von dieser Ansteckreihenfolge zu sein, kann ein Gerät auch mit einer bestimmten Seriennummer geöffnet werden. Das USB-AD14f mit der Seriennummer 157 lässt sich zum Beispiel mit "**usbad14f:@157**" ansprechen.

6.5.1 Eckdaten und Kanalnummern USB-AD14f

Mess-system	Analog	Kanal-nummer	range (Messber.)	range (Aus-gabebereich)	Digital	Richtung
USB-AD14f	16 Eingänge 1 Ausgang	1..16 1	0 (±10.24V)	0 (±5.12V)	2 Ports (je 8 Bit)	1: Eingang (Bit 0..7) 2: Ausg. (Bit 0..7)

Der erste analoge Eingangskanal eines USB-AD14f beginnt bei 1. Damit ergeben sich für die 16 Analogeingänge folgende Konstanten:

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)
```

 © BMC Messsysteme GmbH

Der analoge Ausgangskanal eines USB-AD14f erhält die folgende Konstante:

C

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

Die Richtung der digitalen Portleitungen ist nicht umschaltbar. Dabei stehen die 8 (USB-AD14f) Leitungen des ersten Ports (DIO1) auf Eingang, die 8 (USB-AD14f) Leitungen des zweiten Ports (DIO2) auf Ausgang. Für die Kanäle ergeben sich folgende Konstanten:

C

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

Bei dem Digitaleingang wird die Eingangsleitung 1 gleichzeitig auch als Zähler benutzt. Der Zähler wird mit der folgenden Kanalkonstanten angesprochen:

C

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

6.6 USB-AD16f / AMS42/84-USB

Um ein USB-Messsystem vom Typ USB-AD16f und AMS42/84-USB mit der LIBAD4 zu öffnen, muss an ad_open**()** der String "**usbbase**" übergeben werden. Mehrere USB-AD16f oder AMS42/84-USB Messsysteme lassen sich durch Angabe der Gerätenummer öffnen (1. Gerät mit "**usbbase:0**", 2. Gerät mit "**usbbase:1**", usw.). Die Reihenfolge der Geräte wird durch das Anstecken bestimmt.

Da USB Messsysteme im Betrieb an- und abgesteckt werden können, ist es möglich, dass die Gerätenummern nicht aufeinander folgend vergeben sind. Steckt man z. B. drei USB-AD16f oder AMS42/84-USB an und entfernt dann das 2. Gerät, sind die verbleibenden Geräte mit "**usbbase:0**" und "**usbbase:2**" anzusprechen.

Um unabhängig von dieser Ansteckreihenfolge zu sein, kann ein Gerät auch mit einer bestimmten Seriennummer geöffnet werden. Das Gerät mit der Seriennummer 157 lässt sich zum Beispiel mit "**usbbase:@157**" ansprechen.

6.6.1 Eckdaten und Kanalnummern USB-AD16f / AMS42/84-USB

Mess-system	Analog	Kanal-nummer	range (Messber.)	range (Ausgabebereich)	Digital	Richtung
USB-AD16f / AMS42/84-USB	16 Eingänge 2 Ausgänge	1..16 1 .. 2	0 (±1.024V) 1 (±2.048V) 2 (±5.12V) 3 (±10.24V)	0 (±10.24V)	2 Ports (je 4 Bit)	1: Eingang (Bit 0..3) 2: Ausg. (Bit 0..3)

Die 16 Analogeingänge eines USB-AD16f oder AMS42/84-USB besitzen die Kanalnummern 1-16. Die beiden Analogausgänge haben die Kanalnummern 1 und 2.

Damit ergeben sich für die 16 Analogeingänge folgende Konstanten:

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)
```

Die beiden analogen Ausgangskanäle eines USB-AD16f oder AMS42/84-USB erhalten die folgenden Konstanten:

C

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

Die Richtung der digitalen Portleitungen ist nicht umschaltbar. Dabei stehen die 4 Leitungen des ersten Ports (DIO1) auf Eingang, die 4 Leitungen des zweiten Ports (DIO2) auf Ausgang. Für die Kanäle ergeben sich folgende Konstanten:

C

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

Außerdem besitzt das USB-AD16f und AMS42/84-USB einen Zählereingang, für den sich die folgende Konstante ergibt:

C

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

6.7 USB-OI16

Um ein USB-Messsystem vom Typ USB-OI16 mit der LIBAD4 zu öffnen, muss an ad_open**()** der String "**usb-oi16**" übergeben werden. Mehrere USB-OI16 Geräte lassen sich durch Angabe der Gerätenummer öffnen (1. Gerät mit "**usb-oi16:0**", 2. Gerät mit "**usb-oi16:1**", usw.). Die Reihenfolge der Geräte wird durch das Anstecken bestimmt.

Da USB Geräte im Betrieb an- und abgesteckt werden können, ist es möglich, dass die Gerätenummern nicht aufeinander folgend vergeben sind. Steckt man z. B. drei USB-OI16 an und entfernt dann das 2. Gerät, sind die verbleibenden USB-OI16 mit "**usb-oi16:0**" und "**usb-oi16:2**" anzusprechen.

Um unabhängig von dieser Ansteckreihenfolge zu sein, kann ein Gerät auch mit einer bestimmten Seriennummer geöffnet werden. Das Gerät mit der Seriennummer 157 lässt sich zum Beispiel mit "**usb-oi16:@157**" ansprechen.

6.7.1 Eckdaten und Kanalnummern USB-OI16

Messsystem	Digital	Kanalnummer
USB-OI16	2 Ports (je 16 Bit)	1: Eingang 2: Ausgang

© BMC Messsysteme GmbH

6.7.2 Kanalnummern USB-OI16

Die USB-OI16 stellt zwei 16-Bit Digitalports zur Verfügung. Die Richtung der digitalen Portleitungen ist nicht umschaltbar. Dabei stehen die 16 Leitungen des ersten Ports (DIO1) auf Eingang, die 16 Leitungen des zweiten Ports (DIO2) auf Ausgang. Es ergeben sich folgende Konstanten:

C

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

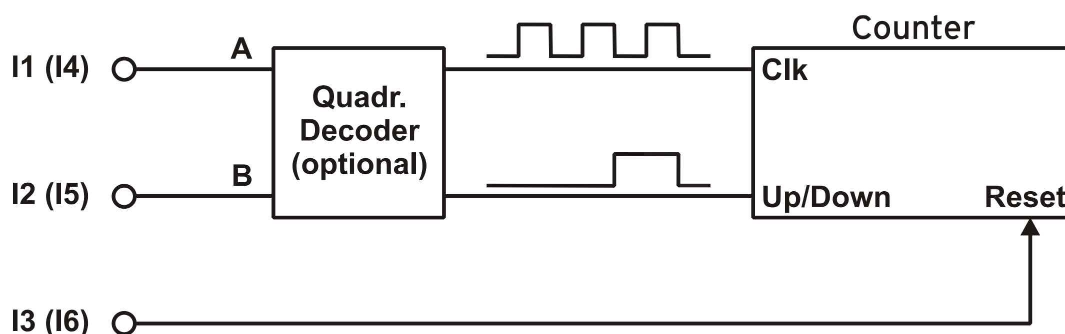
Außerdem besitzt die USB-OI16 zwei 32-Bit Zählereingänge. Diese können in verschiedenen Betriebsarten verwendet werden und müssen vor der Verwendung per Software konfiguriert werden (siehe "Konfiguration der USB-OI16 Zähler", S. 64). Die Eingänge der Zähler (Signal A, Signal B, Reset) werden mit den ersten Digitaleingangsleitungen der USB-OI16 verbunden.

Für die 32-Bit Zählereingänge ergeben sich folgende Konstanten:

C

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

6.7.3 Konfiguration der USB-OI16 Zähler



Zur Einstellung der Zähler werden die Konfigurationsparameter in die Struktur **ad_counter_mode** eingetragen und an **ad_ioctl()** übergeben.

Das folgende Beispiel zeigt das prinzipielle Vorgehen: Es konfiguriert den ersten Zähler des USB-OI16 in der Betriebsart "Zähler".

Prototype

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

© BMC Messsysteme GmbH

C

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

struct ad_counter_mode par;
int32_t adh;
int32_t st;

...

  adh = ad_open ("pcibase");

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

Die Elemente der Struktur haben folgende Bedeutung:

cha

Legt den Zählerkanal fest, der konfiguriert werden soll.

mode

Legt die Betriebsart des Zählers fest.

Betriebsart	Beschreibung
AD_CNT_COUNTER	Der Zählerkanal arbeitet als einfacher Zähler. Es wird nur der Eingang A des Zählers verwendet, wobei jede positive Flanke am Eingang den Zählerstand um eins erhöht.
AD_CNT_UPDOWN	Der Zählerkanal arbeitet als Up/Down Zähler, also mit umschaltbarer Zählrichtung. Dabei arbeitet der Eingang A des Zählers als Takteingang, der Eingang B übernimmt die Richtungsumschaltung. Steht der Eingang B des Zählers auf low, dann erhöht jede positive Flanke am Eingang A den Zählerstand um eins. Ansonsten erniedrigt die positive Flanke den Zählerstand.
AD_CNT_QUAD_DECODER	Der Zähler decodiert an den Eingängen A und B die zwei Spuren eines Inkrementalgebers. Dabei wird jede Flanke der beiden Spuren dekodiert.
AD_CNT_PULSE_TIME	Konfiguriert den Zähler für die Pulsweitenmessung. Dabei ist der Zählereingang mit einer internen Taktquelle (24 MHz) verbunden und wird mit jeder Flanke am Eingang A gestartet und gestoppt.

mux_a, mux_b, mux_rst

Legt die Pins der beiden Digitalports fest, die mit den jeweiligen Eingängen des Zählers verbunden sind. Dabei ist es nicht möglich, die Eingänge eines Zählers mit verschiedenen Digitalports zu verbinden (d. h. die Eingänge A, B und *Reset* müssen entweder alle mit Pins aus dem Port A verbunden sein oder alle mit Pins aus dem Port B).

`mux_a`, `mux_b` **oder** `mux_rst`	**Port/Line**	`mux_a`, `mux_b` **oder** `mux_rst`	**Port/Line**

© BMC Messsysteme GmbH

0	PA/1
1	PA/2
2	PA/3
3	PA/4
4	PA/5
5	PA/6
6	PA/7
7	PA/8
8	PA/9
9	PA/10
10	PA/11
11	PA/12
12	PA/13
13	PA/14
14	PA/15
15	PA/16

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

flags

Legt die Arbeitsweise der Zählereingänge fest. Die Arbeitsweisen können mit **OR** verknüpft werden, z. B. **AD_CNT_INV_RST|AD_CNT_ENABLE_RST**.

Arbeitsweise	Beschreibung
AD_CNT_INV_A	Zählereingang A reagiert invertiert.
AD_CNT_INV_B	Zählereingang A reagiert invertiert.
AD_CNT_INV_RST	Reseteingang reagiert invertiert.
AD_CNT_ENABLE_RST	Reseteingang ist aktiviert.

© BMC Messsysteme GmbH

7 Index

6

64-Bit ... 1

A

Abtastrate ... 25, 30
Abtasttakt ... 23, 34
Abtastzeit ... 22, 43
ad_analog_in () ... 18
ad_analog_out () ... 19
ad_calc_run_size () ... 43
ad_close () ... 8
ad_digital_in () ... 19
ad_digital_out () ... 19
ad_discrete_in () ... 10
ad_discrete_in64 () ... 11
ad_discrete_inv () ... 12
ad_discrete_out () ... 13
ad_discrete_out64 () ... 14
ad_discrete_outv () ... 15
ad_float_to_sample () ... 18
ad_get_digital_line () ... 20
ad_get_drv_version () ... 21
ad_get_line_direction () ... 20
ad_get_next_run () ... 44
ad_get_next_run_f () ... 44
ad_get_next_run_f64 () ... 45
ad_get_product_info () ... 21
ad_get_range_count () ... 8
ad_get_range_info () ... 9
ad_get_sample_layout () ... 40
ad_get_samples () ... 41
ad_get_samples_f () ... 41
ad_get_samples_f64 () ... 42
ad_get_version () ... 20
ad_open () ... 4, 6
ad_poll_scan_state () ... 45
ad_sample_to_float () ... 16, 17
ad_sample_to_float64 () ... 18
ad_set_digital_line () ... 19
ad_set_line_direction () ... 20
ad_start_mem_scan () ... 39
ad_start_scan () ... 40
ad_stop_scan () ... 45
AMS42-LAN16f ... 46
AMS42-LAN16fx ... 46
AMS42-USB ... 62
AMS84-LAN16f ... 46
AMS84-LAN16fx ... 46
AMS84-USB ... 62
Analogausgang
  mehrere setzen ... 15
  setzen ... 13, 14
Anzahl der Messwerte ... 22, 23, 25, 26, 30
Ausgaberichtung ... 20
Ausgang
  mehrere setzen ... 15
  setzen ... 13, 14
Ausgangsbereich ... 13, 14, 15, 46
Ausgangsleitung ... 20
Auslesen der Messwerte ... 29

B

Buffer ... 5, 22, 25, 29, 43
buffer_start ... 40
bytes_per_run ... 25, 43

C

cha ... 22, 49, 53, 57, 58, 65
chac ... 57
chav ... 57

D

Digitalkanal
  Richtung abfragen ... 20
  Richtung setzen ... 20

E

Effektivwert ... 23
Eingaberichtung ... 20
Eingänge
  Reihenfolge ... 39, 40
Eingangsleitung ... 20
Einzelwert
  abfragen ... 10, 11
  mehrere abfragen ... 12
Ergebnis ... 45

F

Fehlernummer ... 4, 6, 45
Fenstertrigger ... 24
Firmwareversion ... 21
flags ... 25, 26, 50, 54, 66
Flankentrigger ... 24
Funktionsgenerator ... 57

G

GetLastError ... 4, 6
Groß-/Kleinschreibung ... 4, 6

H

Headerdatei ... 4

I

Inkrementalgeber 50, 54, 66
Internetadresse 1

K

Kanalart 46
Kanalnummer 10, 11, 12, 13, 14, 15, 22, 46
kontinuierliche Messung 25, 30

L

LAN-AD16f 46
LAN-AD16fx 46
  Digitalports 47
  Kanalnummer 46
  Zähler 47, 48

M

MADDA16 55
MADDA16n 55
Maximum 23
MDA
  Funktionsgenerator 57
MDA12 56
MDA12-4 56
MDA16 56
MDA16-2i 56
MDA16-4i 56
MDA16-8i 56
memory-only Messung 27, 39, 44, 45
Messbereich 10, 11, 12, 23, 46
  Anzahl 8
  Information 9
Messbereichsgrenze 10, 44
Messbereichsmitte 10, 11
Messdaten-Speicherverwaltung
  intern 22, 25, 34, 40, 41, 42, 45
Messsystem
  mehrere gleiche öffnen 7
  mehrere verschiedene öffnen 4, 6
  Name 4
  öffnen 4, 6
  schließen 4, 8
Messung
  kontinuierlich 22, 30
  memory-only 22, 27
  mit Trigger 34
Messwert 13, 14, 16, 17, 18, 44
  auslesen 35
Minimum 23
Mittelwert 23
mode 49, 53, 65
mux_a 50, 54, 66
mux_b 50, 54, 66
mux_rst 50, 54, 66

N

Nachgeschichte 25, 34, 37
Name 4
network byte order 44
Nullpegel 23

O

oder-Operator (|) 46

P

PCI-BASE1000 51
PCI-BASE300 51
PCI-BASEII 51
  Digitalports 51
  Zähler 52
PCIe-BASE 51
  Digitalports 51
  Zähler 52
PCIe-Karten
  Seriennummer 51
PCI-PIO 51
  Digitalports 51
  Zähler 52
Periodenmessung 50, 66
posthist 25, 26
posthist_samples 41
Posthistory 34
prehist 25
prehist_samples 41
Prehistory 34
Produktinformation 21
Produktname 21

Q

Quadraturdekoder 50, 54, 66

R

ram 57
range 22, 58
rate 58
ratio 23
Richtung 20
RMS 23
RUN 26, 30, 32, 33, 34, 40, 43
runs_pending 26

S

sample_rate 25, 43
samples_per_run 23, 25, 43
Scan 5, 22
  erster Messwert 40
  Parameter 22
  starten 28
  stoppen 29
  Zustand 35
Scanparameter 22
Seriennummer 21, 51, 60, 61, 62, 64
Spannungswert 16, 17, 18, 29
Speicherart 23
Speicherintervall 23

© BMC Messsysteme GmbH

Speicherrate ... 34
start ... 40
start_addr ... 58
Starten des Scans ... 28
stop_addr ... 58
Stoppen des Scans ... 29, 45
store ... 23
struct ad_scan_cha_desc ... 22
struct ad_scan_desc ... 25
struct ad_scan_state ... 26

T

ticks_per_run ... 25, 43
Treiberversion ... 21
trg_mode ... 23
trg_par ... 23
Trigger ... 23, 24, 25, 26, 34
  analog ... 35
  Bedingung ... 24, 35
  digital ... 36
  Einstellungen ... 22
  Fenster ... 24
  Flanke ... 24
  Parameter ... 23
  Parameter der Scankanäle ... 35
  positive Flanke ... 35

U

Überlauf der Messwerte ... 22, 30, 45
Umrechnung
  Messwert in Spannungswert ... 16, 17, 18
  Spannungswert Messwert ... 18
Up/Down Zähler ... 50, 54, 66
Urheberrechte ... 2
USB-AD ... 60
  Digitalports ... 60
  Kanalnummer ... 60
  Reihenfolge ... 60
  Seriennummer ... 60
USB-AD12f ... 61
  Digitalports ... 62
  Kanalnummer ... 61
  Reihenfolge ... 61
  Seriennummer ... 61
USB-AD14f ... 61
  Digitalports ... 62
  Kanalnummer ... 61
  Reihenfolge ... 61
  Seriennummer ... 61
USB-AD16f ... 62
  Digitalports ... 63
  Kanalnummer ... 63
  Reihenfolge ... 62
  Seriennummer ... 62
USB-AD-OEM
  Digitalports ... 60
  Kanalnummer ... 60
USB-OI16 ... 64
  Digitalports ... 64
  Kanalnummer ... 64
  Reihenfolge ... 64
  Seriennummer ... 64
  Zähler ... 64, 65
USB-PIO ... 60
  Digitalports ... 61
  Kanalnummer ... 61
  Reihenfolge ... 60
  Richtung ... 61
  Seriennummer ... 60
USB-PIO-OEM ... 60
  Digitalports ... 61
  Kanalnummer ... 61
  Reihenfolge ... 60
  Richtung ... 61
  Seriennummer ... 60

V

Version
  LIBAD4.DLL ... 20
  Treiber ... 21
Vorgeschichte ... 25, 34, 37
  Anzahl Messwerte ... 41
  erster Messwert ... 40

Z

Zähler ... 50, 54, 66
  Konfiguration ... 48, 52, 65
Zeiger ... 29, 39
zero ... 23
Zustand der Messung ... 26, 45