

Programmierhandbuch

Version 4

► w w w . b m c m . d e

LIBAD4

Library for Programming Interface
Libad

Inhaltsverzeichnis

1 Überblick	7
1.1 Einleitung	7
1.2 BMC Messsysteme GmbH	8
1.3 Urheberrechte	9
2 Installation	10
2.1 Installation unter Windows®	10
2.2 Installation unter Mac OS X	10
2.3 Installation unter FreeBSD	11
2.4 Installation unter Linux	13
2.5 Weitergabe der Bibliothek	15
3 Grundlagen	16
3.1 Einführung	16
4 Einzelwerterfassung	18
4.1 Funktionsbeschreibung (Einzelwerte)	18
4.1.1 ad_open	18
4.1.2 ad_close	21
4.1.3 ad_discrete_in	22
4.1.4 ad_discrete_in64	23
4.1.5 ad_discrete_inv	24
4.1.6 ad_discrete_out	25
4.1.7 ad_discrete_out64	26
4.1.8 ad_discrete_outv	28
4.1.9 ad_sample_to_float	29
4.1.10 ad_sample_to_float64	30
4.1.11 ad_float_to_sample	30
4.1.12 ad_float_to_sample64	31
4.1.13 ad_analog_in	32

4.1.14	ad_analog_out	33
4.1.15	ad_digital_in	33
4.1.16	ad_digital_out	33
4.1.17	ad_set_digital_line	34
4.1.18	ad_get_digital_line	34
4.1.19	ad_get_line_direction	35
4.1.20	ad_set_line_direction	35
4.1.21	ad_get_version	36
4.1.22	ad_get_drv_version	36

5 Scanvorgang 37

5.1	Einführung	37
5.2	Scanparameter	37
5.2.1	struct ad_scan_cha_desc	37
5.2.1.1	Speichermöglichkeiten	39
5.2.1.2	Triggermöglichkeiten	40
5.2.2	struct ad_scan_desc	41
5.2.3	struct ad_scan_state	42
5.3	CAN	43
5.3.1	Scanparameter	43
5.3.2	Kanalnummerierung	45
5.4	Memory-only Scan	45
5.4.1	Starten eines Scans	45
5.4.2	Auslesen der Messwerte	47
5.4.3	Stoppen des Scans	48
5.5	Kontinuierliche Messung	49
5.5.1	Aufbau eines RUNs	49
5.5.2	Ein Messwert pro RUN	51
5.5.3	Signale mit unterschiedlicher Speicherrate	53
5.6	Funktionsbeschreibung (Scan)	55
5.6.1	ad_start_mem_scan	55
5.6.2	ad_start_scan	56
5.6.3	ad_calc_run_size	57
5.6.4	ad_get_next_run	58
5.6.5	ad_get_next_run_f	59
5.6.6	ad_poll_scan_state	60
5.6.7	ad_stop_scan	60

6 Messsysteme	61
6.1 iM-AD25a / iM-AD25 / iM3250T / iM3250	61
6.1.1 Kanalnummern iM-AD25a / iM-AD25	62
6.1.2 Kanalnummern iM3250T	63
6.1.3 Kanalnummern iM3250	63
6.2 PCI-BASE300/1000	64
6.2.1 MAD12/12a/12f/16/16a/16f	64
6.2.2 MDA12/12-4/16	65
6.2.3 MCAN	67
6.2.4 Digitalports	67
6.3 PC16TR / PC20TR	67
6.4 PC20NHDL / PC20NVL / P1000TR/ P1000NV	69
6.5 PIO24II / PIO48II	70
6.6 meM-AD /-ADDA /-ADf / -ADfo	71
6.7 meM-PIO / meM-PIO-OEM	73
6.8 USB-AD / USB-PIO	74
6.8.1 Eckdaten und Kanalnummern USB-AD	76
6.8.2 Eckdaten und Kanalnummern USB-PIO	77
7 Index	78

1 Überblick

1.1 Einleitung

Die Bibliothek **LIBAD4** ist eine Schnittstelle zu allen Messsystemen der BMC Messsysteme GmbH. Diese Schnittstelle erlaubt 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 aus diesem Grund zeitlich von der Applikation entkoppelt. Damit ist es möglich, schnell und ohne Verlust von Messwerten die Eingangskanäle abzutasten.

Die **LIBAD4** liegt sowohl für Windows® 2000/XP, als auch für Mac OS X, FreeBSD und Linux vor. Damit ist es ohne Änderung des Sourcecodes möglich, Messsysteme der BMC Messsysteme GmbH Plattform übergreifend einzusetzen.



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>.



► w w w . b m c m . d e

1.3 Urheberrechte

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 © 2007

Stand: 13. Juni 2007

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 unter Windows®



Unter Windows® ist die **LIBAD4** als "dynamic link library" realisiert. Das Installationsprogramm kopiert die Bibliothek inklusive aller Headerfiles und den Beispielpogrammen auf die Festplatte.

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

2.2 Installation unter Mac OS X



Unter Mac OS X ist die **LIBAD4** als "dynamic library" realisiert. Das Installationsprogramm kopiert die Bibliothek inklusive aller Headerfiles und den Beispielpogrammen auf die Festplatte. Die Standardeinstellung kopiert sämtliche Dateien in das Verzeichnis **/Macintosh HD/Developer/SDKs** und kann während der Installation geändert werden.

Nach Abschluss des Installers befinden sich folgende Dateien, wie nachfolgend abgebildet, auf der Festplatte (s. Abbildung 1).

Damit die Programme auf die dynamic library zugreifen können, muss diese in ein Verzeichnis kopiert werden, in dem der dynamic linker shared libraries erwartet. Genaue Hinweise dazu entnehmen Sie bitte der manpage von **dyld**.

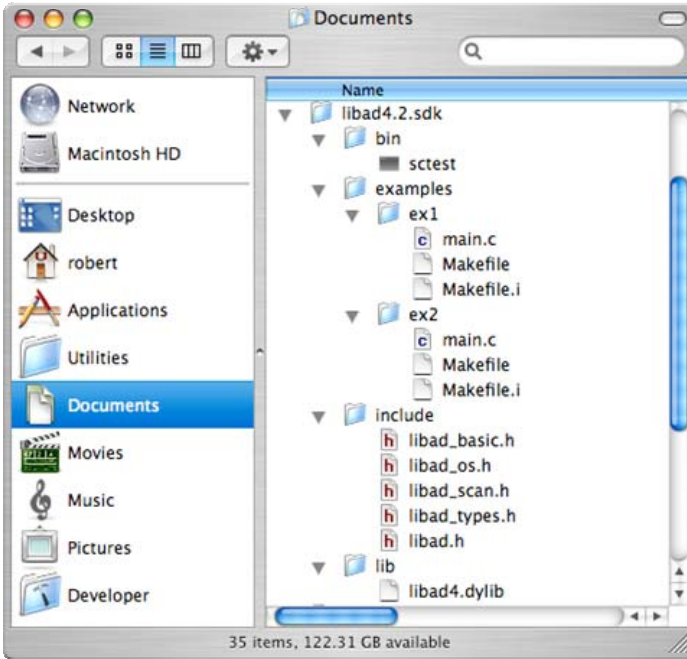


Abbildung 1

Folgender Befehl kopiert die **LIBAD4** nach `/usr/local/lib`:

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

2.3 Installation unter FreeBSD



Unter FreeBSD wird die **LIBAD4** als gepacktes TAR File ausgeliefert. Das File kann mit folgendem Befehl ausgepackt werden (bitte passen Sie die Versionsnummer an die Version der verwendeten LIBAD an).

```
bash# tar xjf libad-freebsd-4.1.333.tar.bz2
bash#
```

Nach dem Auspacken befinden sich folgende Dateien auf der Festplatte:

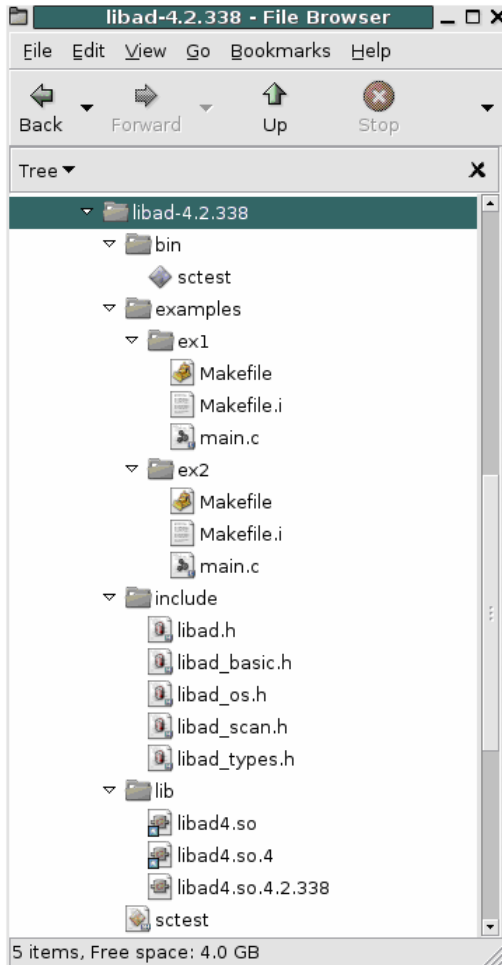


Abbildung 2

Unter FreeBSD ist die **LIBAD4** als "shared library" realisiert. Damit die Programme auf die Bibliothek zugreifen können, muss diese in ein Verzeichnis kopiert werden, in dem der dynamic linker shared libraries erwartet. Genaue Hinweise dazu entnehmen Sie bitte der manpage von **ldconfig** bzw. **ld-elf.so.1**.

Ist Ihr System so eingerichtet, dass shared libraries in **/usr/local/lib** berücksichtigt werden, dann kopieren Sie bitte **libad4.so.4.1.333** nach **/usr/local/lib**. Legen Sie dann zwei symbolische Links **/usr/local/lib/libad4.so.4** und **/usr/local/lib/libad4.so** an, so dass diese auf **/usr/local/lib/libad4.so.4.1.333** zeigen (die Versionsnummer Ihrer **LIBAD4** ist eventuell eine andere und muss entsprechend angepasst werden).

Folgende Befehle führen die notwendigen Aktionen aus:

```
bash# cp lib/libad4.so.4.1.333 /usr/local/lib/libad4.so.4.1.333
bash# ln -sf libad4.so.4.1.333 /usr/local/lib/libad4.so.4
bash# ln -sf libad4.so.4.1.333 /usr/local/lib/libad4.so
bash# /sbin/ldconfig
bash#
```

2.4 Installation unter Linux



Unter Linux wird die **LIBAD4** als gepacktes TAR File ausgeliefert. Das File kann mit folgendem Befehl ausgepackt werden (bitte passen Sie die Versionsnummer an die Version der verwendeten **LIBAD** an).

```
bash# tar xjf libad-linux-4.1.333.tar.bz2
bash#
```

Nach dem Auspacken befinden sich folgende Dateien auf der Festplatte:

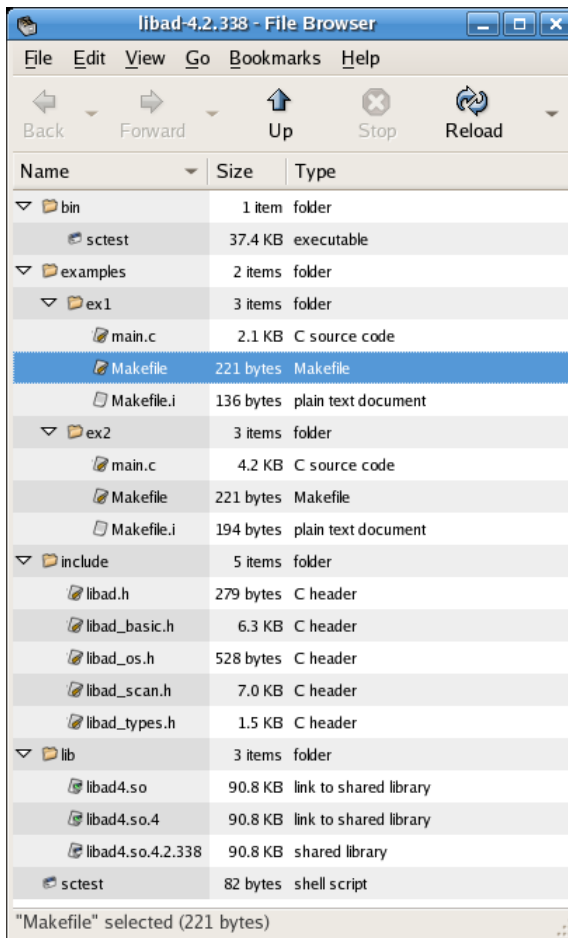


Abbildung 3

Unter Linux ist die **LIBAD4** als "shared library" realisiert. Damit die Programme auf die Bibliothek zugreifen können, muss diese in ein Verzeichnis kopiert werden, in dem **ldconfig** shared libraries erwartet. Genaue Hinweise dazu entnehmen Sie bitte der manpage von **ldconfig** bzw. der Datei **/etc/ld.so.conf**.

Ist Ihr System so eingerichtet, dass shared libraries in **/usr/local/lib** berücksichtigt werden, dann kopieren Sie bitte **libad4.so.4.1.333** nach **/usr/local/lib**. Legen Sie dann zwei symbolische Links

`/usr/local/lib/libad4.so.4` und `/usr/local/lib/libad4.so` an, so dass diese auf `/usr/local/lib/libad4.so.4.1.333` zeigen (die Versionsnummer Ihrer **LIBAD4** ist eventuell eine andere und muss entsprechend angepasst werden). Folgende Befehle führen die notwendigen Aktionen aus:

```
bash# cp lib/libad4.so.4.1.333 /usr/local/lib/libad4.so.4.1.333
bash# ln -sf libad4.so.4.1.333 /usr/local/lib/libad4.so.4
bash# ln -sf libad4.so.4.1.333 /usr/local/lib/libad4.so
bash# /sbin/ldconfig
bash#
```

2.5 Weitergabe der Bibliothek

Damit eine Applikation auf die Funktionen der LIBAD Bibliothek zugreifen kann, muss diese 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
libad4.dylib
libad4.so.4.1.333
```

Es ist Aufgabe des Installationsprogramms der erstellten Applikation, das entsprechende File zusammen mit der Applikation zu installieren. Auf keinen Fall sollte der LIBAD SDK verwendet werden, um die LIBAD Bibliothek auf dem Zielrechner zu installieren.



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

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 **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:



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

<pre>#include "libad.h" ... int32_t adh; ... adh = ad_open ("usb-ad"); if (adh == -1) { printf ("failed to open USB-AD driver\n"); exit (1); } ... ad_close (adh);</pre>

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 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 18).

Die unterstützten Messsysteme sind in Kapitel 6 "Messsysteme" ab Seite 61 beschrieben. Dort sind auch die benötigten Kanalnummer für die Ein- und Ausgangskanäle und die entsprechenden Messbereiche definiert.

Sobald ein Messsystem geöffnet worden ist, lassen sich Messwerte von den Eingängen einlesen (siehe "**ad_discrete_in()**", Seite 22) oder die Werte für die Ausgänge festlegen (siehe "**ad_discrete_out()**", Seite 25). Genaue Hinweise dazu entnehmen Sie bitte dem Kapitel "Einzelwerterfassung".

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 Programmierung eines Scans ist im Kapitel "Scanvorgang" ab Seite 37 beschrieben.

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

C	<pre>#include "libad.h" ... int32_t adh; ... adh = ad_open ("usb-ad"); if (adh == -1) { printf ("failed to open USB-AD\n"); exit (1); } ... ad_close (adh);</pre>
---	--

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. "pci300" und "Pci300" öffnen beide die PCI-BASE300/1000. 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. Folgendes Beispiel öffnet ein USB-AD und eine USB-PIO:



```
C
#include "libad.h"

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

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

...

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

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
#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 Messsysteme über seine Seriennummer öffnen. Dabei ist die Seriennummer mit einem @ Zeichen nach dem Doppelpunkt anzugeben. Folgendes Beispiel öffnet die zwei USB-AD Geräte mit den Seriennummern 157 und 158.



```
C
#include "libad.h"

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

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

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

4.1.2 ad_close



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

C	<pre>#include "libad.h" ... int32_t adh; ... adh = ad_open ("usb-ad"); if (adh == -1) { printf ("failed to open USB-AD\n"); exit (1); } ... ad_close (adh);</pre>
---	---

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

4.1.3 ad_discrete_in



Prototype	<pre>int32_t ad_discrete_in (int32_t adh, int32_t cha, int32_t range, uint32_t *data);</pre>
C	<pre>int32_t adh; int32_t st; uint32_t data; ... adh = ad_open ("usb-ad"); st = ad_discrete_in (adh, AD_CHA_TYPE_ANALOG_IN 1, 0, &data) ... ad_close (adh);</pre>

Die Funktion **ad_discrete_in()** 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_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 ist abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (s. "Messsysteme", S. 61).

4.1.4 ad_discrete_in64



Prototype

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

C

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

...

adh = ad_open ("usb-adh");

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 (z.B. CAN) 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_float64()** zur Verfügung. Die Hilfsfunktion **ad_analog_in()** übergibt den Messwert direkt als Spannung.

Die Kanalnummer und die Nummer des Messbereichs ist abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (s. "Messsysteme", S. 61).

4.1.5 ad_discrete_inv



Prototype

```
int32_t
ad_discrete_inv (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 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.

4.1.6 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 Messbereich 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 Messbereich 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 Messbereichs ist abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (s. "Messsysteme", S. 61).

4.1.7 ad_discrete_out64



Prototype

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

C

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

...

adh = ad_open ("pci300");

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

...

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

...

ad_close (adh);
```

Die Funktion **ad_discrete_out64()** setzt einen Ausgang. Neben der Kanalnummer wird der Funktion noch ein Messbereich 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 Messbereich mit den Hardwareeinstellungen übereinstimmt.

Wie beim analogen Eingangskanal entspricht der Wert **0x0000000000000000** eines Analogausgangs der niedrigsten Ausgangsspannung, der Wert **0x1000000000000000** der höchsten Ausgangsspannung. Die vollen 64Bit von **ad_discrete_out64()** werden wie bei den Eingängen nur von speziellen 64-Bit Messsystemen (z.B. CAN) 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 Messbereichs ist abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (s. "Messsysteme", S. 61).

4.1.8 ad_discrete_outv



Prototype	<pre>int32_t ad_discrete_outv (int32_t adh, int32_t chac, int32_t chav[], uint64_t rangev[], uint64_t datav[]);</pre>
C	<pre>#define CHAC 3 uint64_t rangev[CHAC], datav[CHAC]; int32_t chav[CHAC], adh, i; /* das Beispiel setzt die 3 Digitalports der USB-PIO * auf die Werte 1, 2 und 4 */ adh = ad_open ("usb-pio"); if (adh < 0) { fprintf (stderr, "error: couldn't open USB-PIO\n"); return -1; } /* setze den range bei allen Kanälen auf 0 */ memset (rangev, 0, sizeof(*rangev)); for (i = 0; i < CHAC; i++) { /* Kanalnummer setzen */ chav[i] = AD_CHA_TYPE_DIGITAL_IO (i+1); /* auf Ausgang setzen */ ad_set_line_direction (adh, chav[i], 0); /* Wert setzen */ datav[i] = 1 << i; } ad_discrete_outv (adh, CHAC, chav, rangev, datav); ad_close (adh);</pre>

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 Messbereiche 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, Messbereiche und Werte in Feldern. Die Feldwerte müssen wie in **ad_discrete_out64()** gesetzt werden.

4.1.9 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);
```

C

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

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

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 (s. "Messsysteme", S. 61).

4.1.10 ad_sample_to_float64



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

C	<pre>int32_t adh; int32_t st, cha, range; uint64_t data; float volt; ... adh = ad_open ("usb-ad"); ... cha = AD_CHA_TYPE_ANALOG_IN 1; range = 0; st = ad_discrete_in64 (adh, cha, range, &data); if (st == 0) st = ad_sample_to_float (adh, cha, range, data, &volt); ... ad_close (adh);</pre>
----------	---

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 (s. "Messsysteme", S. 61).

4.1.11 ad_float_to_sample

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 (s. "Messsysteme", S. 61).



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);
```

4.1.12 ad_float_to_sample64

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 (s. "Messsysteme", S. 61).



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);
```

4.1.13 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 ist abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (s. "Messsysteme", S. 61).

4.1.14 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 ist abhängig von der eingesetzten Messhardware und in den entsprechenden Kapiteln dokumentiert (s. "Messsysteme", S. 61).

4.1.15 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.

4.1.16 ad_digital_out

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



Prototype	<pre>int32_t ad_digital_out (int32_t adh, int32_t cha, uint32_t data);</pre>
------------------	--

4.1.17 ad_set_digital_line



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

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 null, wird die Leitung gesetzt. Die erste Leitung in einem Digitalkanal hat die Nummer 0.

4.1.18 ad_get_digital_line



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

Diese Hilfsfunktion liest den Kanal **AD_CHA_TYPE_DIGITAL_IO|cha** und setzt dann ***data** 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.

4.1.19 ad_get_line_direction



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

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.

4.1.20 ad_set_line_direction



Prototype	<pre>int32_t ad_set_line_direction (int32_t adh, int32_t cha, int32_t mask);</pre>
------------------	--

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 ersten Leitung des Digitalports fest.

Je nach Messsystem kann eventuell nicht jede Leitung einzeln in der Richtung umgeschaltet werden oder die Richtung ist fest eingestellt (z. B. der Digitalport der PCI-BASE300/1000).

4.1.21 **ad_get_version**



Prototype	<code>uint32_t ad_get_version ();</code>
------------------	--

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.

4.1.22 **ad_get_drv_version**



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

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

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 blockweise an den Aufrufer ab. Der Aufrufer ist in diesem Fall dafür verantwortlich die Blöcke 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      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 hardwareabhängig und in den entsprechenden Abschnitten des Kapitels "Messsysteme", S. 61 beschrieben.
- **range**
Legt den Messbereich des Kanals fest. Die Nummer des Messbereichs ist Hardware abhängig und in den entsprechenden Abschnitten des Kapitels "Messsysteme", S. 61 beschrieben.
- **store**
Legt zusammen mit **ratio** (s. u.) fest, wie der Kanal gespeichert wird. Eine ausführliche Beschreibung der Speicherarten folgt im nächsten Abschnitt (s. "Speichermöglichkeiten", S. 39).
- **ratio**
Legt das Speicherintervall fest (s. "Speichermöglichkeiten", S. 39).
- **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[]** (s. u.) fest, ob und wie dieser Kanal einen Trigger auslösen soll.

➤ **trg_par[]**

Legen 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 Speichermöglichkeiten

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	24ms	...
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	a ₁₃	...
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	a ₁₃	...

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

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).

Zeit	0ms	2ms	4ms	6ms	8ms	10ms	12ms	14ms	16ms	18ms	20ms	22ms	24ms	...
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	a ₁₃	...
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	24ms	...
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	a ₁₃	...
Speicherung					a ₅ $\frac{1}{5}\sum a_i$					a ₁₀ $\frac{1}{5}\sum a_i$...

5.2.1.2 Triggermöglichkeiten

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()**.

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

```

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 an das aufrufende Programm abgegeben werden.
- **bytes_per_run**
Wird von **LIBAD4** zurückgegeben, legt die Größe des Buffers für **ad_get_next_run()** fest (in Bytes).

➤ **samples_per_run**

Wird von **LIBAD4** zurückgegeben, legt die Anzahl der Messwerte eines Buffers fest, der von **ad_get_next_run_f()** zurückgegeben wird.



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

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

Die Elemente der Struktur haben folgende Bedeutung:

➤ **flags**

Zeigt den Zustand der Messung an (s. 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**

Liefert die Anzahl der RUNs, die zum Auslesen bereit sind.

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
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.

5.3 CAN

Eine Ausnahme bei der Speicherung bilden CAN Messgeräte. Hier werden die CAN Messages in einem internen Speicher abgelegt und dieser wird anstelle der Kanäle abgetastet. Damit wird eine äquidistante Abtastung der CAN Signale realisiert und diese fügen sich nahtlos in die analogen Werte ein.

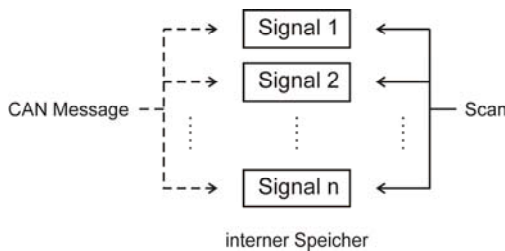


Abbildung 5

5.3.1 Scanparameter

Die Parameter zur Definition eines CAN Signals werden mit der Hilfsfunktion **ad_set_can_cha()** eingestellt.

C	<pre>void ad_set_can_cha (struct ad_scan_cha_desc *cha, int bus, int off, int len, int nbo, int sgn, uint32_t id, int moff, int mlen);</pre>
----------	--

Setzt die Struktur **struct ad_scan_cha_desc** für die angegebenen Parameter.

- **bus**
Legt die Busnummer des Signals fest.
- **off**
Legt den Offset des Signals in der Message fest.
- **len**
Legt die Anzahl der Bits des Signals in der Message fest. Dabei ist das erste Bit das mit **off** festgelegte.
- **nbo**
Legt fest ob das Signal in der "network byte order" angeordnet ist. Dabei bedeutet **nbo != 0** "network byte order".
- **sgn**
Legt fest ob das Signal "signed" oder "unsigned" ist. Dabei bedeutet **sgn != 0** "signed".
- **id**
Legt die Messagenummer des Signals fest.
- **moff**
Legt den Multiplexoffset des Signals fest.
- **mlen**
Legt die Anzahl der Bits des Multiplexoffsets fest. Dabei ist das erste Bit das mit **moff** festgelegte.

Die übrigen Scanparameter wie zum Beispiel **ratio**, **store** oder **trg_mode** funktionieren wie bei analogen oder digitalen Messsystemen.

5.3.2 Kanalnummerierung

Die Kanalnummerierung in einem Scan mit CAN Signalen entspricht der Reihenfolge, in der die Signale dem Scan hinzugefügt werden. Das n-te hinzugefügte Signal erscheint damit als Kanalnummer n. Diese Kanäle existieren rein virtuell.

5.4 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.

5.4.1 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
    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_cha_desc** gesetzt werden. Folgendes Beispiel setzt die Abtastrate auf 1kHz und speichert 500 Messwerte (pro Kanal).



```
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
    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).

5.4.2 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 16Bit-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
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 **a_i** kommen vom Analogeingang 1, die Messwerte **b_i** von Analogeingang 3):

Feldindex	0	1	2	...	498	499	500	501	502	...	998	999
Zeit	0ms	1ms	2ms	...	498ms	499ms	0ms	1ms	2ms	...	498ms	499ms
Messwert	a₁	a₂	a₃	...	a₄₉₉	a₅₀₀	b₁	b₂	b₃	...	b₄₉₉	b₅₀₀

5.4.3 Stoppen des Scans

Jeder Scanvorgang muss gestoppt werden, sobald der Aufruf der Funktion **ad_start_scan()** als Ergebnis **0** zurückgeliefert hat.



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:



```
C
int32_t scan_result;
...

rc = ad_stop_scan (adh, &scan_result);

...
```


5.5 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. Damit ist es möglich jeden einzelnen Messwert sofort nach der Abtastung zu erhalten. Allerdings lassen sich mit dieser Einstellung selbstverständlich nur noch 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.

5.5.1 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	0ms	1ms	2ms	...	248ms	249ms	0ms	1ms	2ms	...	248ms	249ms
Messwert	a₁	a₂	a₃	...	a₂₄₉	a₂₅₀	b₁	b₂	b₃	...	b₂₄₉	b₂₅₀

RUN #0

Feldindex	0	1	2	...	248	249	250	251	252	...	498	499
Zeit	250ms	251ms	252ms	...	498ms	499ms	250ms	251ms	252ms	...	498ms	499ms
Messwert	a₂₅₁	a₂₅₂	a₂₅₃	...	a₄₉₉	a₅₀₀	b₂₅₁	b₂₅₂	b₂₅₃	...	b₄₉₉	b₅₀₀

RUN #1

Folgender Beispielcode liest die RUNs während der Messung aus:



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);
...

```

5.5.2 Ein Messwert pro RUN

Wird `ticks_per_run` auf 1 gestellt, dann werden RUNs mit einem Messwert pro Signal erzeugt:



```

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;

...
    
```

Das obige Beispiel erzeugt 500 RUNs mit folgendem Aufbau:

Feldindex	0	1
Zeit	0ms	0ms
Spannungswert	a_1	b_1

RUN #0

Feldindex	0	1
Zeit	10ms	10ms
Spannungswert	a_2	b_2

RUN #1

Feldindex	0	1
Zeit	4980ms	4980ms
Spannungswert	a ₄₉₉	b ₄₉₉

RUN #498

Feldindex	0	1
Zeit	4990ms	4990ms
Spannungswert	a ₅₀₀	b ₅₀₀

RUN #499

5.5.3 Signale mit unterschiedlicher Speicherrate

Die beiden obigen Beispiele 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.

Vereinfachend wurde bisher der Wert **ticks_per_run** der Struktur **struct ad_scan_desc** als die Anzahl der Messwerte pro Signal beschrieben. Diese Vereinfachung ist dann zulässig, wenn alle Signale im Verhältnis 1:1 gespeichert werden. In diesem Fall ist die Anzahl der gespeicherten Messwerte pro Signal identisch mit der Anzahl der Abtasttakte einer Messung.

Wird ein Signal in einem anderen Verhältnis als 1:1 gespeichert, dann muss genau zwischen Abtasttakt (*ticks*), der Anzahl der Samples pro Signal und den Abtasttakten pro RUN unterschieden werden. Folgendes Diagramm stellt einen Scan dar, in dem zwei Eingangskanäle abgetastet und gespeichert werden. Der Scan läuft mit einer Abtastrate von 2ms (50Hz), der Eingang **a** wird 1:1 gespeichert, der Eingang **b** speichert den Mittelwert über 5 Messwerte.

Zeit	0ms	2ms	4ms	6ms	8ms	10ms	12ms	14ms	16ms	18ms	20ms	22ms	24ms	...
Erfassung	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	...
Speicherung	a ₁	a ₂	a ₃	a ₄	$\frac{1}{5} \sum b_i$	a ₆	a ₇	a ₈	a ₉	$\frac{1}{5} \sum b_i$	a ₁₁	a ₁₂	a ₁₃	...

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	3	4	5	6
Zeit	0ms	2ms	4ms	6ms	8ms	8ms
Spannungswert	a₁	a₂	a₃	a₄	a₅	$\frac{1}{5} \sum b_i$

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	0ms	2ms	4ms	...	496ms	498ms	8ms	18ms	28ms	...	488ms	498ms
Spannungswert	a₁	a₂	a₃	...	a₂₄₉	a₂₅₀	$\frac{1}{5} \sum b_i$	$\frac{1}{5} \sum b_i$	$\frac{1}{5} \sum b_i$...	$\frac{1}{5} \sum b_i$	$\frac{1}{5} \sum b_i$

5.6 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 wird ein Zeiger auf ein Element der Struktur **struct ad_scan_desc** übergeben und die Zahl der abzutasten-

den 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 Digitaleingänge abgetastet, müssen erst alle analogen und dann 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 (s. "ad_calc_run_size", S. 57).

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);
```

Im Gegensatz zu `ad_start_mem_scan()` wertet `ad_start_scan()` das Element `ticks_per_run` der Struktur **struct ad_scan_desc** aus. Damit lässt sich ein Scan auf mehrere RUNs verteilen (s. "Kontinuierliche Messung", S. 49).



Aufgrund von Einschränkungen bei den meisten Messkarten müssen die Eingangskanäle unbedingt in aufsteigender Reihenfolge im Feld `chav[]` angegeben werden! Werden außer Analogeingängen auch Digitaleingänge abgetastet, müssen erst alle analogen und dann 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 (s. "ad_calc_run_size", S. 57).

5.6.3 ad_calc_run_size



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

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

```
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));
...
```

5.6.4 ad_get_next_run



Prototype

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

Liefert die Messwerte eines Scans.

Die Funktion `ad_get_next_run()` liefert die Messwerte direkt vom Messsystem (also als 16Bit Werte), die untere Messbereichsgrenze entspricht dem Wert `0x0000`, die obere Messbereichsgrenze dem Wert `0xffff` (genauer gesagt entspricht die obere Grenze dem Wert `0x10000`, 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).

5.6.5 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.

ad_get_next_run_f() liefert Messwerte als Float-Werte, die untere Messbereichsgrenze entspricht dem Wert **0x0000**, die obere Messbereichsgrenze dem Wert **0xffff** (genauer gesagt entspricht die obere Grenze dem Wert **0x10000**, der nicht erreicht wird), die bereits (je nach Messbereich) in die zugehörigen Spannungswerte umgerechnet sind.



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).

5.6.6 ad_poll_scan_state



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

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.

5.6.7 ad_stop_scan



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

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

6 Messsysteme

Ein- bzw. Ausgangskanäle werden in **LIBAD4** durch Kanalnummern spezifiziert. Die Kanalnummer (Integer mit 32Bit) legt neben der eigentlichen Nummer des Kanals auch noch 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:

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

Die verwendeten Kanalnummern sind abhängig vom eingesetzten Messsystem und in den entsprechenden Kapiteln dokumentiert. Beispielsweise lässt sich der erste Analogeingang einer PCI-BASE300/1000 angeben durch den Ausdruck **AD_CHA_TYPE_ANALOG_IN | 1**.

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

6.1 iM-AD25a / iM-AD25 / iM3250T / iM3250



Um ein iM-AD25a, iM-AD25, iM3250T oder iM2350 mit der **LIBAD4** zu öffnen, muss an **ad_open()** der String "**im:<ip-addr>**" übergeben werden. Dabei muss **<ip-addr>** durch die entsprechende IP-Adresse ersetzt werden. Beispielsweise öffnet der String "**im:192.168.1.1**" das iM-Gerät mit der IP Adresse 192.168.1.1. Beim Öffnen des Treibers wird nicht zwischen den iM-Gerätetypen unterschieden.

Messsystem	Analog	Kanalnummer	Messbereich	range	Digital
iM-AD25a	16 Eingänge	1..16	$\pm 10.24V$ $\pm 5.12V$	1 0	1 Ausg. (4Bit)
iM-AD25	16 Eingänge	1..16	$\pm 5.12V$	0	1 Ausg. (4Bit)
iM3250T	32 Eingänge	17..48	$\pm 5.12V$	0	-
iM3250	32 Eingänge	AnIn 1..16: 1..16 (bei 1 BPL) 17..32 (bei 2 BPL) AnIn 17..32: 33..48	$\pm 5.00V$	0	-



Bitte beachten Sie, dass sich beim iM3250T durch einen eventuell gesteckten MAL-Messverstärker der Messbereich des entsprechenden Kanals verändern kann.

6.1.1 Kanalnummern iM-AD25a / iM-AD25

Der erste analoge Eingangskanal eines iM-AD25a / iM-AD25 beginnt bei 1. Damit ergeben sich für die 16 Analogeingänge folgende Konstanten:

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

Das iM-AD25 verfügt außerdem über einen Digitalausgang mit 4 Leitungen. Dessen Richtung ist nicht umschaltbar.

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

6.1.2 Kanalnummern iM3250T

Der erste analoge Eingangskanal eines iM3250T beginnt bei 17. Damit ergeben sich für die 32 analogen Eingänge folgende Konstanten:

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

6.1.3 Kanalnummern iM3250

Die Kanalnummern des iM3250 hängen von der Ausbaustufe des Geräts ab. Ist nur eine BPL im Gerät vorhanden, erscheinen die ersten 16 Kanäle von 1 bis 16. Falls beide BPLs eingebaut sind, erscheinen die ersten 16 Kanäle von 17 bis 32. Die zweiten 16 Eingänge sind immer unter den Nummern 33 bis 48 erreichbar.

```
#ifdef BPL1    /* 1 bpl installed /

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

#else          /* 2 bpl's installed /

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

#endif /* BPL1 */

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

6.2 PCI-BASE300/1000



Um eine PCI-BASE300/1000 mit der **LIBAD4** zu öffnen, muss an **ad_open()** der String "**pci300**" übergeben werden. Beim Öffnen des Treibers wird nicht zwischen PCI-BASE300 und PCI-BASE1000 unterschieden.

Mehrere Karten lassen sich durch Angabe der Kartennummer unterscheiden (1. Karte mit "**pci300:0**", 2. Karte mit "**pci300:1**", usw.).

6.2.1 MAD12/12a/12f/16/16a/16f

Der erste analoge Eingangskanal eines MAD12/12a/12f/16/16a/16f beginnt bei 1. Sobald ein zweites analoges Eingangsmodul auf der PCI-BASE300/1000 gesteckt ist, wird der erste Eingang des zweiten Moduls unter der Nummer 257 (0x100+1) angesprochen.

Modul	Analog	Kanalnummer	Messbereich	range
MAD12, MAD16	16 Eingänge (single-ended) 8 Eingänge (differentiell)	1..16 (se) 17..24 (diff)	±1.024V ±2.048V ±5.120V ±10.240V 0.06V..5.06V	0 1 2 3 4
MAD12a, MAD12f, MAD16a, MAD16f	16 Eingänge (single-ended) 8 Eingänge (differentiell)	1..16 (se) 17..24 (diff)	±1.024V ±2.048V ±5.120V ±10.240V	0 1 2 3

Damit ergeben sich für die ersten 32 Analogeingänge im single-ended Betrieb folgende Konstanten:



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

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

Sind die Eingangsmodule auf differentiell gejumpert, müssen die Kanalnummern 17..24 verwendet werden. Folgende Konstanten bezeichnen die Kanäle 1..8 des ersten analogen Eingangsmoduls in der differentiellen Betriebsart:



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

Selbstverständlich ist es möglich, ein Eingangsmodul differentiell und das zweite single-ended zu betreiben, so dass dann 24 Eingangskanäle zur Verfügung stehen.

Die Messbereiche der Eingangskanäle sind modulabhängig. Sind zwei verschiedene Eingangsmodule auf die PCI-BASE300/1000 gesteckt, dann können sich die Messbereiche der Kanäle 1..16 von den Messbereichen der Kanäle 17..32 unterscheiden.

6.2.2 MDA12/12-4/16

Ebenso wie beim MAD12/12a/12f/16/16a/16f werden die analogen Ausgänge des zweiten Moduls ab der Nummer 257 (0x100+1) angesprochen.

Modul	Analog	Kanalnummer	Ausgabebereich	range
MDA12, MDA16	2 Ausgänge	1..2	$\pm 10.24V$	0
			$\pm 5.12V$	1
MDA12-4	4 Ausgänge	1..4	$\pm 10.24V$	0
			$\pm 5.12V$	1

Die Ausgabebereiche der Ausgangsmodule MDA12/MDA12-4 und MDA16 werden hardwaremäßig am Modul konfiguriert. Der Aufrufer muss sicherstellen, dass der übergebene Messbereich mit dem konfigurierten Messbereich des Moduls übereinstimmt.

Je nach gesteckten Ausgangsmodulen ergeben sich dann folgende Kanalnummern:



```
#ifndef MDA12 /* first module is a MDA12 (2 chas) /

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

#define AO3 (AD_CHA_TYPE_ANALOG_OUT|0x0101)
#define AO4 (AD_CHA_TYPE_ANALOG_OUT|0x0102)

/* chas 5/6 only if second module is a MDA12-4 */
#define AO5 (AD_CHA_TYPE_ANALOG_OUT|0x0103)
#define AO6 (AD_CHA_TYPE_ANALOG_OUT|0x0104)

#else /* first module is a MDA12-4 (4 chas) /

#define AO1 (AD_CHA_TYPE_ANALOG_OUT|0x0001)
#define AO2 (AD_CHA_TYPE_ANALOG_OUT|0x0002)
#define AO3 (AD_CHA_TYPE_ANALOG_OUT|0x0003)
#define AO4 (AD_CHA_TYPE_ANALOG_OUT|0x0004)

#define AO5 (AD_CHA_TYPE_ANALOG_OUT|0x0101)
#define AO6 (AD_CHA_TYPE_ANALOG_OUT|0x0102)

/* chas 7/8 only if second module is a MDA12-4 */
#define AO7 (AD_CHA_TYPE_ANALOG_OUT|0x0103)
#define AO8 (AD_CHA_TYPE_ANALOG_OUT|0x0104)

#endif /* !MDA12 */
```

6.2.3 MCAN

Das MCAN ist ein CAN-Schnittstellenmodul, das 2 verschiedene CAN-Busse erfassen kann. Ein CAN-Modul kann nur scannen. Es ist nicht möglich diskrete Werte abzuholen.

6.2.4 Digitalports

Die PCI-BASE300/1000 stellt zwei 16Bit-Digitalports zur Verfügung. Die Ports sind in ihrer Richtung fest verdrahtet, der erste Port steht auf Eingang, der zweite Port auf Ausgang. Es wird folgende Nummerierung verwendet:



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

6.3 PC16TR / PC20TR



Um eine PC16TR/PC20TR mit der **LIBAD4** zu öffnen, muss an **ad_open()** der String "**pc20**" übergeben werden. Es wird beim Öffnen des Treibers nicht zwischen PC16TR und PC20TR unterschieden.

Mehrere Karten lassen sich durch Angabe der Kartennummer unterscheiden (1. Karte mit "**pc20:0**", 2. Karte mit "**pc20:1**", usw.).

Mess-system	Analog	Kanal-nummer	range (Messber.)	range (Aus-gabebereich)	Digital	Kanal-nummer
PC20TR	16 Eing. 2 Ausg.	1..16 1 .. 2	0 ($\pm 10V$) 1 ($\pm 5V$) 2 ($\pm 2V$) 3 ($\pm 1V$)	0 ($\pm 10V$) 1 ($\pm 5V$)	2 Ports (je 16Bit)	1..2
PC16TR	16 Eing.	1..16	0 ($\pm 10V$)	-	2 Ports (je 8Bit)	1..2

Der erste analoge Eingangskanal einer PC16TR/PC20TR beginnt bei 1. Damit ergeben sich für die 16 Analogeingänge folgende Konstanten:



```
#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 Ausgabebereiche der beiden analogen Ausgangskanäle der PC20TR werden hardwaremäßig auf der Messkarte konfiguriert. Der Aufrufer muss sicherstellen, dass der übergebene Messbereich mit dem konfigurierten Messbereich des Ausgangs übereinstimmt. Diese verwenden folgende Kanalnummern:



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

Die Richtung der digitalen Portleitungen ist in 8-er Gruppen umschaltbar (s. "`ad_set_line_direction`"(), S. 35).



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

6.4 PC20NHDL / PC20NVL / P1000TR/ P1000NV



Um eine PC20NHDL/PC20NVL/P1000TR/P1000NV mit der **LIBAD4** zu öffnen, muss an `ad_open()` der String **"p1000"** übergeben werden. Es wird beim Öffnen des Treibers nicht zwischen den einzelnen Kartenversionen unterschieden.

Mehrere Karten lassen sich durch Angabe der Kartennummer öffnen (1. Karte mit **"p1000:0"**, 2. Karte mit **"p1000:1"**, usw.).

Messsystem	Analog	Kanal-nummer	range (Messber.)	range (Ausgabebereich)	Digital	Kanal-nummer
PC20NHDL, PC20NVL, P1000TR, P1000NV	16 Eing. 2 Ausg.	1..16 1 .. 2	0 (±10V) 1 (±5V) 2 (±2V) 3 (±1V)	0 (±10V) 1 (±5V)	2 Ports (je 16Bit)	1..2

Der erste Analogeingang beginnt bei 1. Damit ergeben sich für die 16 analogen Eingänge folgende Konstanten:



```
#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 Ausgabebereiche der beiden analogen Ausgänge werden hardwaremäßig auf der Messkarte konfiguriert. Der Aufrufer muss sicherstellen, dass der übergebene Messbereich mit dem konfigurierten Messbereich des Ausgangs übereinstimmt. Die beiden analogen Ausgänge verwenden folgende Kanalnummern:



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

Die Richtung der digitalen Portleitungen ist in 8-er Gruppen umschaltbar (s. "**ad_set_line_direction**", S. 35).



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

6.5 PIO24II / PIO48II



Um eine PIO24II oder PIO48II mit der **LIBAD4** zu öffnen, muss an **ad_open()** der String "**pioii**" übergeben werden. Es wird beim Öffnen des Treibers nicht zwischen PIO24II und PIO48II unterschieden.

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

Messsystem	Digital	Kanalnummer
PIO48II	6 Ports (je 8Bit)	1..6
PIO24II	3 Ports (je 8Bit)	1..3

Die Richtung der Leitungen ist für jeden Port getrennt einstellbar. Die Umstellung erfolgt portweise (s. "**ad_set_line_direction**", S. 35).



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)
#define DIO4    (AD_CHA_TYPE_DIGITAL_IO|0x0004)
#define DIO5    (AD_CHA_TYPE_DIGITAL_IO|0x0005)
#define DIO6    (AD_CHA_TYPE_DIGITAL_IO|0x0006)
```

6.6 meM-AD /-ADDA /-ADf / -ADfo



Um ein meM-AD/-ADDA/-ADf/-ADfo mit der **LIBAD4** zu öffnen, muss an **ad_open()** der String **"memadusb"** (meM-AD), **"memaddausb"** (meM-ADDA), **"memadfusb"** (meM-ADf) bzw. **"memadfpusb"** (meM-ADfo) übergeben werden. Mehrere USB Messsysteme lassen sich durch Angabe der Gerätenummer öffnen (z. B. 1. Gerät mit **"memadusb:0"**, 2. Gerät mit **"memadusb: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 meM-ADDA an und entfernt dann das 2. Gerät, sind die verbleibenden meM-ADDA mit **"memaddausb:0"** und **"memaddausb: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 **"memadfpusb:@157"** ansprechen.

Messsystem	Analog	Kanal- nummer	Eing./Ausg.- bereich	range	Digital	Kanal- nummer
meM-AD	16 Eingänge	1..16	$\pm 5.12V$	0	-	-
meM-ADDA, meM-ADf	16 Eingänge 1 Ausgang	1..16 1	$\pm 5.12V$	0	2 Ports (je 4Bit)	1: Eingang (Bit 1..4) 2: Ausgang (Bit 1..4)
meM-ADfo	16 Eingänge 1 Ausgang	1..16 1	$\pm 5.12V$	0	2 Ports (je 8Bit)	1..2

Der erste analoge Eingangskanal eines meM-AD/-ADDA/-ADf/-ADfo beginnt bei 1. Damit ergeben sich für die 16 Analogeingänge folgende Konstanten:



```
#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 meM-ADDA/-ADf/-ADfo erhält die folgende Konstante:



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

Die Richtung der Digitalports ist nicht umschaltbar. Dabei stehen die 4 (meM-ADfo: 8) Leitungen des 1. Ports (DIO1) auf Eingang, die 4 (meM-ADfo: 8) Leitungen des 2. Ports (DIO2) auf Ausgang. Es ergeben sich folgende Konstanten:



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


6.7 meM-PIO / meM-PIO-OEM



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

Da die USB Messsysteme im laufenden Betrieb an- und wieder abgesteckt werden können, ist es möglich, dass die Gerätenummern nicht in aufsteigender Reihenfolge vergeben sind. Werden beispielsweise drei Geräte angesteckt und dann das zweite Gerät wieder abgesteckt, sind die beiden verbleibenden Geräte mit **"mempiousb:0"** und **"mempiousb: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 durch Angabe von **"mempiousb:@157"** ansprechen.

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

Die Richtung der Leitungen ist für jeden Port getrennt einstellbar. Die Umstellung erfolgt portweise (s. **"ad_set_line_direction"**, S. 35).



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

6.8 USB-AD / USB-PIO



Um ein USB-AD oder eine USB-PIO 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 Geräteummern 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.



Das USB-AD bzw. die USB-PIO implementiert die CDC Klasse als ACM. Für diese Geräte bietet FreeBSD einen entsprechenden Treiber an, so dass die Geräte direkt von FreeBSD unterstützt werden. Dazu muss der umodem Treiber geladen sein:

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

Um ein USB-AD oder eine USB-PIO mit der **LIBAD4** zu öffnen, muss an **ad_open()** der String **"usb-ad"** bzw. **"usb-pio"** übergeben werden. Unter FreeBSD öffnet die **LIBAD4** daraufhin das Device **"/dev/cuaU0"**, um mit dem USB-AD bzw. USB-PIO zu kommunizieren. Es ist Aufgabe der Applikation sicherzustellen, dass als **"/dev/cuaU0"** ein USB-AD bzw. eine USB-PIO angemeldet ist.

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 **LIBAD4** öffnet daraufhin das Device **"/dev/cuaU0"** und **"/dev/cuaU1"**.

Neben der automatischen Vergabe der Devicenamen lässt sich unter FreeBSD auch das verwendete Device direkt angeben. Beim Aufruf von **ad_open ("usb-ad:/dev/cuaU12"** bzw. **"usb-pio:/dev/cuaU12")** öffnet die **LIBAD4** das Device **"/dev/cuaU12"**.



Das USB-AD bzw. die USB-PIO implementiert die CDC Klasse als ACM. Für diese Geräte bietet Linux einen entsprechenden Treiber an, so dass die Geräte direkt von Linux unterstützt werden, wenn das Kernel entsprechend konfiguriert ist.

Um ein USB-AD oder eine USB-PIO mit der **LIBAD4** zu öffnen, muss an **ad_open()** der String **"usb-ad"** bzw. **"usb-pio"** übergeben werden. Unter Linux öffnet die **LIBAD4** daraufhin das Device **"/dev/ttyACM0"**, um mit dem USB-AD bzw. der USB-PIO zu kommunizieren. Es ist Aufgabe der Applikation sicherzustellen, dass als **"/dev/ttyACM0"** ein USB-AD bzw. eine USB-PIO angemeldet ist.

Mehrere USB Messsysteme lassen sich durch Vergabe des Devicenamens öffnen.

Beim Aufruf von **ad_open ("usb-ad:/dev/ttyACM12"** bzw. **"usb-pio:/dev/ttyACM12")** öffnet die **LIBAD4** das Device **"/dev/ttyACM12"**.

6.8.1 Eckdaten und Kanalnummern USB-AD

Mess-system	Analog	Kanal-nummer	range (Messber.)	range (Aus-gabebereich)	Digital	Richtung
USB-AD	16 Eingänge 1 Ausgang	1..16 1	33 ($\pm 5.12V$)	1 ($\pm 5.12V$)	2 Ports (je 4Bit)	1: Eingang (Bit 1..4) 2: Ausgang (Bit 1..4)

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

```
#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 erhält die folgende Konstante:

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

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:

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

6.8.2 Eckdaten und Kanalnummern USB-PIO

Messsystem	Digital	Kanalnummer
USB-PIO	3 Ports (je 8Bit)	1..3

Die Richtung der Leitungen ist für jeden Port getrennt einstellbar. Die Umstellung erfolgt portweise (s. "**ad_set_line_direction**", S. 35).

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

7 Index

A

Abtastrate 41, 49
Abtasttakt 39, 53, 54
Abtastzeit 37, 57
ad_analog_in () 32
ad_analog_out () 33
ad_calc_run_size () 57
ad_close () 21
ad_digital_in () 33
ad_discrete_in () 22
ad_discrete_in64 () 23
ad_discrete_inv () 25
ad_discrete_out () 25
ad_discrete_out64 () 27
ad_discrete_outv () 28
ad_float_to_sample () 30
ad_float_to_sample64 () 31
ad_get_digital_line () 34
ad_get_drv_version () 36
ad_get_line_direction () 35
ad_get_next_run () 58
ad_get_next_run_f () 59
ad_get_version () 36
ad_open () 19
ad_open () 16
ad_poll_scan_state () 60
ad_sample_to_float () 29
ad_sample_to_float64 () 30
ad_set_can_cha () 43
ad_set_digital_line () 34
ad_set_line_direction () 35
ad_start_mem_scan () 55
ad_start_scan () 56
ad_stop_scan () 60
Analogausgang
 mehrere setzen 28
 setzen 25, 27
Anzahl der Messwerte 37, 39, 42, 49, 53
Ausgabebereich 61
Ausgaberichtung 35
Ausgang

 mehrere setzen 28
 setzen 25, 27
Ausgangsbereich 25, 27
Ausgangsleitung 35
Auslesen der Messwerte 47

B

Bitanzahl 44
Buffer 17, 37, 41, 47, 57
bus 44
Busnummer 44
bytes_per_run 41, 57

C

CAN 43, 67
 Kanalnummerierung 45
 Scanparameter 43
cha 38

D

differentiell 65
Digitalkanal
 Richtung abfragen 35
 Richtung setzen 35

E

Effektivwert 38
Eingaberichtung 35
Eingänge
 Reihenfolge 56, 57
Eingangsleitung 35
Einzelwert
 abfragen 22, 23
 mehrere abfragen 25
Ergebnis 60

F

Fehlernummer 17, 19, 60
flags 42

FreeBSD 7, 11

G

GetLastError 19

Groß-/Kleinschreibung 16, 19

H

Headerdatei 16

I

id 44

iM-3250 61

iM-3250T 61

iM-AD25 61

iM-AD25a 61

Installation

FreeBSD 11

Linux 13

Mac OS X 10

Windows® 10

K

Kanalart 61

Kanalnummer 22, 23, 25, 27, 28, 37, 38,
61

Kanalnummerierung 45

kontinuierliche Messung 41, 49

L

len 44

Linux 7, 13

M

Mac OS X 7, 10

MAD12 64

MAD12a 64

MAD12f 64

MAD16 64

MAD16a 64

MAD16f 64

Maximum 39

MCAN 67

MDA12 65

MDA12-4 65

MDA16 65

meM-AD 71

meM-ADDA 71

meM-Adf 71

meM-ADfo 71

meM-Geräte

Reihenfolge 71, 73

Seriennummer 71, 73

memory-only Messung 45, 55, 59

meM-PIO 73

meM-PIO-OEM 73

Message

Bitanzahl 44

Messagenummer 44

Messbereich 22, 23, 25, 27, 28, 38, 61

Messbereichsgrenze 22, 58, 59

Messbereichsmitte 22, 23

Messsystem

mehrere gleiche öffnen 19

mehrere verschiedene öffnen 17, 19

Name 16

öffnen 16, 19

schließen 16, 21

Messung

kontinuierlich 37, 49

memory-only 37, 45

Messwert 26, 27, 29, 30, 31, 58, 59

Minimum 39

Mittelwert 39

mIen 44

moff 44

Multiplexoffset 44

Bitanzahl 44

N

Nachgeschichte 41

Name 16

nbo 44

network byte order 44

network byte-order 58

Nullpegel 38

O

oder-Operator (!) 61
off 44
Offset 44

P

P1000NV 69
P1000TR 69
PC16TR 67
PC20NHDL 69
PC20NVL 69
PC20TR 67
PCI-BASE
 Digitalports 67
PCI-BASE1000 64
PCI-BASE300 64
PIO24II 70
PIO48II 70
posthist 41, 42
prehist 41

R

range 38
ratio 38
Richtung 35
RMS 38, 39
RUN 42, 49, 51, 53, 56, 57
runs_pending 42

S

sample_rate 41, 57
samples_per_run 39, 42, 57
Scan 17, 37
 starten 45
 stoppen 48
Scanparameter 37
Seriennummer 71, 73, 74
sgn 44
Signal
 signed 44
 unsigned 44
signed 44
single-ended 64

Spannungswert 29, 30, 31, 47
Speicherart 38, 39
Speicherintervall 38
Speicherrate 53
Starten des Scans 45
Stoppen des Scans 48, 60
store 38
struct ad_scan_cha_desc 37
struct ad_scan_desc 41
struct ad_scan_state 42

T

ticks_per_run 41, 57
Treiberversion 36
trg_mode 38
trg_par 39
Trigger 38, 40, 41, 42
Triggerbedingung 40
Triggereinstellungen 37
Triggerparameter 39

U

Überlauf der Messwerte 37, 49, 60
Umrechnung
 Messwert in Spannungswert 29, 30
 Spannungswert Messwert 30
 SpannungswertMesswert 31
unsigned 44
Urheberrechte 9
USB-AD 74
 Kanalnummer 76
 Reihenfolge 74
 Seriennummer 74
USB-PIO 74
 Kanalnummer 77
 Reihenfolge 74
 Richtung 77
 Seriennummer 74

V

Version
 LIBAD4.DLL 36
 Treiber 36
Vorgeschichte 41

W

Windows® 7, 10

Z

Zeiger 47, 55

zero 38

Zustand der Messung 42, 60