Comedi

The *Control and Measurement Device Interface* handbook

David Schleef

ds@schleef.org

Frank Hess

fmhess@users.sourceforge.net

Herman Bruyninckx

Herman.Bruyninckx@mech.kuleuven.ac.be

Abstract

Comedi is a free software project to interface *digital acquisition* (DAQ) cards. It is the combination of three complementary software items: (i) a generic, device-independent API, (ii) a collection of Linux kernel modules that implement this API for a wide range of cards, and (iii) a Linux user space library with a developer-oriented programming interface to configure and use the cards.

1. Overview

Comedi is a *free software* project that develops drivers, tools, and libraries for various forms of *data acquisition*: reading and writing of analog signals; reading and writing of digital inputs/outputs; pulse and frequency counting; pulse generation; reading encoders; etc. The project's source code is distributed in two packages, `comedi (http://www.comedi.org/download.php)` and `comedilib`

(`http://www.comedi.org/download.php`), and provides several Linux *kernel modules* and a *user space* library:

- **Comedi** is a collection of drivers for a variety of common data acquisition plug-in boards (which are called "devices" in Comedi terminology). The drivers are implemented as the combination of (i) one single core Linux kernel module (called "`comedi`") providing common functionality, and (ii) individual low-level driver modules for each device.
- **Comedilib** is a separately distributed package containing a user-space library that provides a developer-friendly interface to the Comedi devices. Included in the *Comedilib* package are documentation, configuration and calibration utilities, and demonstration programs.
- **Kcomedilib** is a Linux kernel module (distributed with the `comedi` package) that provides the same interface as *comedilib* in kernel space, and suitable for *real-time* tasks. It is effectively a "kernel library" for using Comedi from real-time tasks.

Comedi works with standard Linux kernels, but also with its real-time extensions RTAI (http://www.rtai.org) and RTLinux/Free (http://www.fsmlabs.com/products/openrtlinux/).

This section gives a high-level introduction to which functionality you can expect from the software. More technical details and programming examples are given in the following sections of this document.

1.1. What is a "device driver"?

A device driver is a piece of software that interfaces a particular piece of hardware: a printer, a sound card, a motor drive, etc. It translates the primitive, device-dependent commands with which the hardware manufacturer allows you to configure, read and write the electronics of the hardware interface into more abstract and generic function calls and data structures for the application programmer.

David Schleef started the Comedi project to put a generic interface on top of lots of different cards for measurement and control purposes. This type of cards are often called *data acquisition* (or **DAQ**) cards.

Analog input and output cards were the first goal of the project, but now Comedi also provides a device independent interface to digital *input and output* cards, and *counter and timer* cards (including encoders, pulse generators, frequency and pulse timers, etc.).

Schleef designed a structure which is a balance between *modularity* and *complexity*: it's fairly easy to integrate a new card because most of the infrastructure part of other, similar drivers can be reused, and learning the generic and hence somewhat "heavier" Comedi API doesn't scare away new contributors from integrating their drivers into the Comedi framework.

1.2. Policy vs. mechanism

Device drivers are often written by application programmers, that have only their particular application in mind; especially in real-time applications. For example, one writes a driver for the parallel port, because one wants to use it to generate pulses that drive a stepper motor. This approach often leads to device drivers that depend too

much on that particular application, and are not general enough to be re-used for other applications. One golden rule for the device driver writer is to separate mechanism and policy:

- **Mechanism.** The mechanism part of the device interface is a faithful representation of the bare functionality of the device, independent of what part of the functionality an application will use.
- **Policy.** Once a device driver offers a software interface to the mechanism of the device, an application writer can use this mechanism interface to use the device in one particular fashion. That is, some of the data stuctures offered by the mechanism are interpreted in specific physical units, or some of them are taken together because this composition is relevant for the application. For example, a analog output card can be used to generate voltages that are the inputs for the electronic drivers of the motors of a robot; these voltages can be interpreted as setpoints for the desired velocity of these motors, and six of them are taken together to steer one particular robot with six-degrees of freedom. Some of the other outputs of the same physical device can be used by another application program, for example to generate a sine wave that drives a vibration shaker.

So, Comedi focuses only on the *mechanism* part of DAQ interfacing. The project does not provide the policy parts, such as Graphical User Interfaces to program and display acquisitions, signal processing libraries, or control algorithms.

1.3. A general DAQ device driver package

From the point of view of application developers, there are many reasons to welcome the standardization of the API and the architectural structure of DAQ software:

- **API**: devices that offer similar functionalities, should have the same software interface, and their differences should be coped with by parameterizing the interfaces, not by changing the interface for each new device in the family. However, the DAQ manufacturers have never been able (or willing) to come up with such a standardization effort themselves.
- **Architectural structure**: many electronic interfaces have more than one layer of functionality between the hardware and the operating system, and the device driver code should reflect this fact. For example, many different interface cards use the same PCI driver chips, or use the parallel port as an intermediate means to connect to the hardware device. Hence, “lower-level” device drivers for these PCI chips and parallel ports allow for an increased modularity and re-useability of the software. Finding the generic similarities and structure among different cards helps in developing device drivers faster and with better documentation.

In the case of Linux as the host operating system, device driver writers must keep the following Linux-specific issues in mind:

- **Kernel space vs. User space.** The Linux operating system has two levels that require basically different programming approaches. Only privileged processes can run in the kernel, where they have access to all hardware and to all kernel data structures. Normal application programs can run their processes only in user space, where these processes are shielded from each other, and from direct access to hardware and to critical data of the operating system; these user space programs execute much of the operating system’s functionality through *system calls*.

  Device drivers typically must access specific addresses on the bus, and hence must (at least partially) run in kernel space. Normal users program against the API of *Comedi*, while Comedi device driver writers use the API offered by *Kcomedilib*. Typical examples of the latter are the registration of interrupt handler routines, and the handling of events.

- **Device files or device file system.** Users who write an application for a particular device, must link their application to that device's device driver. Part of this device driver, however, runs in kernel space, and the user application in user space. So, the operating system provides an interface between both. In Linux or Unix, these interfaces are in the form of "files" in the `/dev` directory (2.2.x kernels or earlier) or `/devfs` directory (2.4.x kernels and later). Each device supported in the kernel has a representative as such a user space device file, and its functionality can be accessed by classical Unix file I/O: `open`, `close`, `read`, `write`, and `ioctl`.
- **`/proc` interface.** Linux (and some other UNIX operating systems) offer a file-like interface to attached devices (and other OS-related information) via the `/proc` directories. These "files" do not really exist, but it gives a familiar interface to users, with which they can inspect the current status of each device.
- **Direct Memory Access (DMA) vs. Programmed Input/Output (PIO).** Almost all devices can be interfaced in PIO mode: the processor is responsible for directly accessing the bus addresses allocated to the device whenever it needs to read or write data. Some devices also allow DMA: the device and the memory "talk" to each other directly, without needing the processor. DMA is a feature of the bus, not of the operating system (which, of course, has to support its processes to use the feature).
- **Real-time vs. non real-time.** If the device is to be used in a RTLinux/Free (http://www.fsmlabs.com/products/openrtlinux/) or RTAI (http://www.rtai.org) application, there are a few extra requirements, because not all system calls are available in the kernel of the real-time operating systems RTLinux/Free (http://www.fsmlabs.com/products/openrtlinux/) or RTAI (http://www.rtai.org). The APIs of RTAI and RTLinux/Free differ in different ways, so the Comedi developers have spent a lot of efforts to make generic wrappers to the required RTOS primitives: timers, memory allocation, registration of interrupt handlers, etc.

1.4. DAQ signals

The cards supported in Comedi have one or more of the following **signals**: analog input, analog output, digital input, digital output, counter input, counter output, pulse input, pulse output:

- **Digital** signals are conceptually quite simple, and don't need much configuration: the number of channels, their addresses on the bus, and their input or output direction.
- **Analog** signals are a bit more complicated. Typically, an analog acquisition channel can be programmed to generate or read a voltage between a lower and an upper threshold (e.g., `-10V` and `+10V`); the card's electronics can be programmed to automatically sample a set of channels, in a prescribed order, to *buffer* sequences of data on the board; or to use DMA or an interrupt routine to dump the data in a prescribed part of memory.
- **Pulse**-based signals (counters, timers, encoders, etc.) are conceptually only a bit more complex than digital inputs and outputs, in that they only add some *timing specifications* to the signal. Comedi has still only a limited number of drivers for this kind of signals, although most of the necessary API and support functionality is available.

In addition to these "real" DAQ functions, Comedi also offers basic timer access.

1.5. Device hierarchy

Comedi organizes all hardware according to the following generic hierarchy:

- **Channel**: the lowest-level hardware component, that represents the properties of one single data channel; for example, an analog input, or a digital output. Each channel has several parameters, such as: the voltage range; the reference voltage; the channel polarity (unipolar, bipolar); a conversion factor between voltages and physical units; the binary values “0” and “1”; etc.
- **Sub-device**: a set of functionally identical channels that are physically implemented on the same (chip on an) interface card. For example, a set of 16 identical analog outputs. Each sub-device has parameters for: the number of channel and the type of the channels.
- **Device**: a set of sub-devices that are physically implemented on the same interface card; in other words, the interface card itself. For example, the `National Instruments 6024E` device has a sub-device with 16 analog input channels, another sub-device with two analog output channels, and a third sub-device with eight digital inputs/outputs. Each device has parameters for: the device identification tag from the manufacturer, the identification tag given by the operating system (in order to discriminate between multiple interface cards of the same type), the number of sub-devices, etc.

Some interface cards have extra components that don’t fit in the above-mentioned classification, such as an EEPROM to store configuration and board parameters, or calibration inputs. These special components are also classified as “sub-devices” in Comedi.

1.6. Acquisition terminology

This Section introduces the terminology that this document uses when talking about “acquisitions.” Figure 1 depicts a typical acquisition **sequence**:

- The sequence has a **start** and an **end**. At both sides, the software and the hardware need some finite **initialization or settling time**.
- The sequence consists of a number of identically repeated **scans**. This is where the actual data acquisitions are taking place: data is read from the card, or written to it. Each scan also has a **begin**, an **end**, and a finite **setup time**. Possibly, there is also a settling time (“**scan delay**”) at the end of a scan.

  So, the hardware puts a lower boundary (the **scan interval**) on the minimum time needed to complete a full scan.

- Each scan contains one or more **conversions** on particular channels, i.e., the AD/DA converter is activated on each of the programmed channels, and produces a sample, again in a finite **conversion time**, starting from the moment in time called the **sample time** in Figure 1 (sometimes also called the “timestamp”), and caused by a triggering event, called **convert**. In addition, each hardware has limits on the minimum **conversion interval** it can achieve, i.e., the minimum time it needs between *subsequent* conversions.

  Some hardware must *multiplex* the conversions onto one single AD/DA hardware, such that the conversions are done serially in time (as shown on the Figure); other cards have the hardware to do two or more acquisitions in parallel. The begin of each conversion is “triggered” by some internally or externally generated pulse, e.g., a timer.

In general, not only the begin of a *conversion* is triggered, but also the begin of a *scan* and of a *sequence*. Comedi provides the API to configure what triggering source one wants to use in each case. The API also allows to specify the **channel list**, i.e., the sequence of channels that needs to be acquired during each scan.

Figure 1. Acquisition sequence. (Figure courtesy of Kurt Mueller (mailto:Kurt.Mueller@sfwte.ch).)

1.7. DAQ functions

The basic data acquisition functionalities that Comedi offers work on channels, or sets of channels:

- **Single acquisition**: Comedi has function calls to synchronously perform *one single* data acquisition on a specified channel: `comedi_data_read()`, `comedi_data_write()`, `comedi_dio_read()`, `comedi_dio_write()`. "Synchronous" means that the calling process blocks until the data acquisition has finished.
- **Instruction**: a `comedi_do_insn()` instruction performs (possibly multiple) data acquisitions on a specified channel, in a **synchronous** way. So, the function call blocks until the whole acquisition has finished.

  In addition, `comedi_do_insnlist()` executes a *list* of instructions (on different channels) in one single (blocking, synchronous) call, such that the overhead involved in configuring each individual acquisition is reduced.

- **Scan**: a scan is an acquisition on a set of different channels, with a *specified sequence and timing*.

  Scans are not directly available as stand-alone function calls in the Comedi API. They are the internal building blocks of a Comedi *command* (see below).

- **Command**: a command is *sequence* of *scans*, for which conditions have been specified that determine when the acquisition will start and stop. A `comedi_command()` function call generates **aynchronous** data acquisition: as soon as the command information has been filled in, the `comedi_command()` function call returns, the hardware of the card takes care of the sequencing and the timing of the data acquisition, and makes sure that the acquired data is delivered in a software buffer provided by the calling process. Asynchronous operation requires some form of "callback" functionality to prevent buffer overflow: after the calling process has launched the acquisition command, it goes off doing other things, but not after it has configured the "handler" that the interface card can use when it needs to put data in the calling process's buffer. Interrupt routines or DMA are typical techniques to allow such asynchronous operation. Their handlers are configured at driver load time, and can typically not be altered from user space.

  Buffer management is not the only asynchronous activity: a running acquisition must eventually be stopped too, or it must be started after the `comedi_command()` function call has prepared (but not started) the hardware for the acquisition. The command functionality is very configurable with respect to choosing which **events** will signal the starting or stopping of the programmed acquisition: external triggers, internal triggers, end of scan interrupts, timers, etc. The user of the driver can execute a Comedi *instruction* that sends a trigger signal to the device driver. What the driver does exactly with this trigger signal is determined in the specific

driver. For example, it starts or stops the ongoing acquisition. The execution of the event associated with this trigger instruction is **synchronous** with the execution of the trigger instruction in the device driver, but it is **asynchronous** with respect to the instruction or command that initiated the current acquisition.

Typically, there is one synchronous triggering instruction for each *subdevice*.

Note that software triggering is only relevant for commands, and not for instructions: instructions are executed *synchronously* in the sense that the instruction call blocks until the whole instruction has finished. The command call, on the other hand, activates an acquisition and returns before this acquisition has finished. So, the software trigger works asynchronously for the ongoing acquisition.

1.8. Supporting functionality

The full command functionality cannot be offered by DAQ cards that lack the hardware to autonomously sequence a series of scans, and/or to support interrupt or DMA callback functionality. For these cards, the command functionality must be provided in software. And because of the quite strict real-time requirements for a command acquisition, a real-time operating system should be used to translate the command specification into a correctly timed sequence of instructions. Such a correct translation is the responsibility of the device driver developer for the card. However, Comedi provides the `comedi_rt_timer` kernel module to support such a **virtual command execution** under RTAI or RTLinux/Free.

Comedi not only offers the API **to access** the functionality of the cards, but also **to query** the capabilities of the installed devices. That is, a user process can find out *on-line* what channels are available, and what their physical parameters are (range, direction of input/output, etc.).

Buffering is another important aspect of device drivers: the acquired data has to be stored in such buffers, because, in general, the application program cannot guarantee to always be ready to provide or accept data as soon as the interface board wants to do a read or write operation. Therefore, Comedi offers all functionality to configure and manage data buffers, abstracting away the intricacies of buffer management at the bare operating system level.

As already mentioned before, Comedi contains more than just procedural function calls, since it also offers **event-driven** (“asynchronous”) functionality: the data acquisition can signal its completion by means of an interrupt or a *callback* function call. Callbacks are also used to signal errors during the data acquisition or when writing to buffers, or at the end of a scan or acquisition that has been launched previously to take place asynchronously (i.e., the card fills up som shared memory buffer autonomously, and only warns the user program after it has finished). The mechanisms for synchronization and interrupt handling are a bit different when used in real-time (RTAI or RTLinux/Free) or non real-time, but both contexts are encapsulated wihting the same Comedi calls.

Because multiple devices can all be active at the same time, Comedi provides **locking** primitives to ensure atomic operations on critical sections of the code or data structures.

Finally, Comedi offers the previously mentioned “high-level” interaction, i.e., at the level of user space device drivers, through file operations on entries in the `/dev` directory (for access to the device’s functionality), or

interactively from the command line through the “files” in the `/proc` directory (which allow to inspect the status of a Comedi device).

2. Configuration

This section assumes that you have successfully compiled and installed the Comedi software, that your hardware device is in your computer, and that you know the relevant details about it, i.e., what kind of card it is, the I/O base, the IRQ, jumper settings related to input ranges, etc.

2.1. Configuration

Before being able to get information from a DAQ card, you first have to tell the Comedi core kernel module which device you have, which driver you want to attach to the card, and which run-time options you want to give to the driver. This configuration is done by running the **comedi_config** command. (As root of course.) Here is an example of how to use the command (perhaps you should read its **man** page now):

```
PATH=/sbin:/usr/sbin:/usr/local/sbin:$PATH
comedi_config /dev/comedi0 labpc-1200 0x260,3
```

This command says that the “file” `/dev/comedi0` can be used to access the Comedi device that uses the *`labpc-1200`* board, and that you give it two run-time parameters (`0x260` and `3`). More parameters are possible, for example to discriminate between two or more identical cards in your system.

If you want to have the board configured in this way every time you boot, put the lines above into a start-up script file of your Linux system (for example, the `/etc/rc.d/rc.local` file), or for PCMCIA boards the appropriate place is the /etc/pcmcia/comedi script. For non-PCMCIA boards, you can also arrange to have your driver loaded and comedi_config run with by adding a few lines to /etc/modules.conf (see the INSTALL file for the comedi kernel modules). You can, of course, also run comedi_config at a command prompt.

This tutorial goes through the process of configuring Comedi for two devices, a `National Instruments AT-MIO-16E-10`, and a `Data Translation DT2821-F-8DI`.

The NI board is plug-and-play. The current ni_atmio driver has kernel-level ISAPNP support, which is used by default if you do not specify a base address. So you could simply run comedi_config as

```
comedi_config /dev/comedi0 ni_atmio
```

For the `Data Translation` board, you need to have a list of the jumper settings; these are given in the Comedi manual section about this card. (Check first to see whether they are still correct!) The card discussed her is a `DT2821-f-8di`. The **man** page of **comedi_config** tells you that you need to know the I/O base, IRQ, DMA 1, DMA 2. However, the Comedi driver also recognizes the differential/single-ended and unipolar/bipolar jumpers. As always, the source is the final authority, and looking in `module/dt282x.c` tells us that the options list is interpreted as:

(... TO BE FILLED IN ...)

So, the appropriate options list is:

```
0x200,4„1,1,1
```

and the full configuration command is:

```
comedi_config /dev/comedi1 dt2821-f-8di 0x200,4„1,1,1
```

The differential/single-ended number is left blank, since the driver already knowns (from the board name), that it is differential. Also the DMA numbers are left blank, since we don't want the driver to use DMA. (Which could interfere with the sound card...) Keep in mind that things commented in the source, but not in the documentation are about as likely to change as the weather, so put good comments next to the following line when you put it in a start-up file.

So now you have your boards configured correctly. Since data acquisition boards are not typically well-engineered, Comedi sometimes can't figure out if the board is actually there. If it can't, it assumes you are right. Both of these boards are well-made, so Comedi will give an error message if it can't find them. The Comedi kernel module, since it is a part of the kernel, prints messages to the kernel logs, which you can access through the command **dmesg** or the file `/var/log/messages`. Here is a configuration failure (from **dmesg**):

```
comedi0: ni_atmio: 0x0200 can't find board
```

When it does work, you get:

```
comedi0: ni_atmio: 0x0260 at-mio-16e-10 ( irq = 3 )
```

Note that it also correctly identified the board.

2.2. Getting information about a card

So now that you have Comedi talking to the hardware, try to talk to Comedi. Here's some pretty low-level information, which can sometimes be useful for debugging:

```
cat /proc/comedi
```

On the particular system this demonstration was carried out, this command gives:

```
comedi version 0.6.4
format string
 0: ni_atmio             at-mio-16e-10           7
 1: dt282x               dt2821-f-8di            4
```

This documentation feature is not well-developed yet. Basically, it currently returns the driver name, the device name, and the number of subdevices.

In the `demo/` directory, there is a command called **info**, which provides information about each subdevice on the board. Its output can be rather long, if the board has several subdevices. Here's part of the output of the

`National Instruments` board (which is on `/dev/comedi0`), as a result of the command **demo/info /dev/comedi0**:

```
overall info:
  version code: 0x000604
  driver name: ni_atmio
  board name: at-mio-16e-10
  number of subdevices: 7
subdevice 0:
  type: 1 (analog input)
  number of channels: 16
  max data value: 4095
...
```

The overall info gives information about the device; basically the same information as `/proc/comedi`.

This board has seven subdevices. Devices are separated into subdevices that each have a distinct purpose; e.g., analog input, analog output, digital input/output. This board also has an EEPROM and calibration DACs that are also subdevices.

Comedi has more information about the device than what is displayed above, but **demo/info** doesn't currently display this.

3. Writing Comedi programs

This Section describes how a well-installed and configured Comedi package can be used in an application, to communicate data with a set of Comedi devices. Section 4 gives more details about the various acquisition functions with which the application programmer can perform data acquisition in Comedi.

Also don't forget to take a good look at the `demo` directory of the Comedilib source code. It contains lots of examples for the basic functionalities of Comedi.

3.1. Your first Comedi program

This example requires a card that has analog or digital input. This progam opens the device, gets the data, and prints it out:

```
#include <stdio.h>   /* for printf() */
#include <comedilib.h>

int subdev = 0;         /* change this to your input subdevice */
int chan = 0;           /* change this to your channel */
int range = 0;          /* more on this later */
int aref = AREF_GROUND; /* more on this later */

int main(int argc,char *argv[])
{
```

```
  comedi_t *it;
  lsampl_t data;

  it=comedi_open("/dev/comedi0");

  comedi_data_read(it,subdev,chan,range,aref, & data);

  printf("%d\n",data);

  return 0;
}
```

The `comedi_open()` can only be successful if the `comedi0` device file is configured to point to a valid Comedi driver. Section 2.1 explains how this driver is linked to the “device file”.

The code above is basically the guts of `demo/inp.c`, without error checking or fancy options. Compile the program using

```
cc tut1.c -lcomedi -o tut1
```

(Replace `cc` by your favourite C compiler command.)

The *`range`* variable tells Comedi which gain to use when measuring an analog voltage. Since we don’t know (yet) which numbers are valid, or what each means, we’ll use `0`, because it won’t cause errors. Likewise with *`aref`*, which determines the analog reference used.

3.2. Converting samples to voltages

If you selected an analog input subdevice, you probably noticed that the output of **tut1** is a number between `0` and `4095`, or `0` and `65535`, depending on the number of bits in the A/D converter. Comedi samples are *always* unsigned, with `0` representing the lowest voltage of the ADC, and `4095` the highest. Comedi compensates for anything else the manual for your device says. However, you probably prefer to have this number translated to a voltage. Naturally, as a good programmer, your first question is: “How do I do this in a device-independent manner?”

Most devices give you a choice of gain and unipolar/bipolar input, and Comedi allows you to select which of these to use. This parameter is called the “range parameter,” since it specifies the “input range” for analog input (or “output range” for analog output.) The range parameter represents both the gain and the unipolar/bipolar aspects.

Comedi keeps the number of available ranges and the largest sample value for each subdevice/channel combination. (Some devices allow different input/output ranges for different channels in a subdevice.)

The largest sample value can be found using the function

```
 lsampl_t comedi_get_maxdata(comedi_t * device, unsigned int subdevice, unsigned int channel))
```

The number of available ranges can be found using the function:

```
int comedi_get_n_ranges(comedi_t * device, unsigned int subdevice, unsigned int channel);
```

For each value of the range parameter for a particular subdevice/channel, you can get range information using:

```
comedi_range * comedi_get_range(comedi_t * device,
                                unsigned int subdevice, unsigned int channel, unsigned int range
```

which returns a pointer to a comedi_range structure, which has the following contents:

```
typedef struct{
        double min;
        double max;
        unsigned int unit;
}comedi_range;
```

The structure element *`min`* represents the voltage corresponding to comedi_data_read() returning `0`, and *`max`* represents comedi_data_read() returning *`maxdata`*, (i.e., `4095` for `12` bit A/C converters, `65535` for `16` bit, or, `1` for digital input; more on this in a bit.) The *`unit`* entry tells you if *`min`* and *`max`* refer to voltage, current, or are dimensionless (e.g., for digital I/O).

“Could it get easier?” you say. Well, yes. Use the function `comedi_to_phys()` comedi_to_phys(), which converts data values to physical units. Call it using something like

```
volts=comedi_to_phys(it,data,range,maxdata);
```

and the opposite

```
data=comedi_from_phys(it,volts,range,maxdata);
```

3.3. Using the file interface

In addition to providing low level routines for data access, the Comedi library provides higher-level access, much like the standard C library provides `fopen()`, etc. as a high-level (and portable) alternative to the direct UNIX system calls `open()`, etc. Similarily to `fopen()`, we have comedi_open():

```
file=comedi_open("/dev/comedi0");
```

where *`file`* is of type *`(comedi_t *)`*. This function calls `open()`, as done explicitly in a previous section, but also fills the comedi_t structure with lots of goodies; this information will be useful soon.

Specifically, you need to know *`maxdata`* for a specific subdevice/channel. How about:

```
maxdata=comedi_get_maxdata(file,subdevice,channel);
```

Wow! How easy. And the range information?

```
comedi_range * comedi_get_range(comedi_tcomedi_t *it,unsigned int subdevice,unsigned int chan,uns
```

3.4. Your second Comedi program: simple acquisition

Actually, this is the first Comedi program again, just that we've added what we've learned.

```
#include <stdio.h>      /* for printf() */
#include <comedilib.h>

int subdev = 0;         /* change this to your input subdevice */
int chan = 0;           /* change this to your channel */
int range = 0;          /* more on this later */
int aref = 0;           /* more on this later */

int main(int argc,char *argv[])
{
  comedi_t *cf;
  int chan=0;
  lsampl_t data;
  int maxdata,rangetype;
  double volts;

  cf=comedi_open("/dev/comedi0");

  maxdata=comedi_get_maxdata(cf,subdev,chan);

  rangetype=comedi_get_rangetype(cf,subdev,chan);

  comedi_data_read(cf->fd,subdev,chan,range,aref,&data);

  volts=comedi_to_phys(data,rangetype,range,maxdata);

  printf("%d %g\n",data,volts);

 return 0;
}
```

3.5. Your third Comedi program: instructions

This program (taken from the set of demonstration examples that come with Comedi) shows how to use a somewhat more flexible acquisition function, the so-called instruction.

```
#include <stdio.h>
#include <comedilib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/time.h>
#include <unistd.h>
#include "examples.h"

/*
```

```
 * This example does 3 instructions in one system call.  It does
 * a gettimeofday() call, then reads N_SAMPLES samples from an
 * analog input, and the another gettimeofday() call.
 */

#define MAX_SAMPLES 128

comedi_t *device;

int main(int argc, char *argv[])
{
  int ret,i;
  comedi_insn insn[3];
  comedi_insnlist il;
  struct timeval t1,t2;
  lsampl_t data[MAX_SAMPLES];

  parse_options(argc,argv);


  device=comedi_open(filename);
  if(!device){
    comedi_perror(filename);
    exit(0);
  }

  if(verbose){
    printf("measuring device=%s subdevice=%d channel=%d range=%d analog reference=%d\n",
      filename,subdevice,channel,range,aref);
  }

  /* Set up a the "instruction list", which is just a pointer
   * to the array of instructions and the number of instructions.
   */
  il.n_insns=3;
  il.insns=insn;

  /* Instruction 0: perform a gettimeofday() */
  insn[0].insn=INSN_GTOD;
  insn[0].n=2;
  insn[0].data=(void *)&t1;

  /* Instruction 1: do 10 analog input reads */
  insn[1].insn=INSN_READ;
  insn[1].n=n_scan;
  insn[1].data=data;
  insn[1].subdev=subdevice;
  insn[1].chanspec=CR_PACK(channel,range,aref);

  /* Instruction 2: perform a gettimeofday() */
  insn[2].insn=INSN_GTOD;
  insn[2].n=2;
  insn[2].data=(void *)&t2;

  ret=comedi_do_insnlist(device,&il);
  if(ret<0){
    comedi_perror(filename);
```

```
    exit(0);
  }

  printf("initial time: %ld.%06ld\n",t1.tv_sec,t1.tv_usec);
  for(i=0;i<n_scan;i++){
    printf("%d\n",data[i]);
  }
  printf("final time: %ld.%06ld\n",t2.tv_sec,t2.tv_usec);

  printf("difference (us): %ld\n",(t2.tv_sec-t1.tv_sec)*1000000+
      (t2.tv_usec-t1.tv_usec));

  return 0;
}
```

3.6. Your fourth Comedi program: commands

This example programs an analog output subdevice with Comedi's most powerful acquisition function, the asynchronous command, to generate a waveform.

The waveform in this example is a sine wave, but this can be easily changed to make a generic function generator.

The function generation algorithm is the same as what is typically used in digital function generators. A 32-bit accumulator is incremented by a phase factor, which is the amount (in radians) that the generator advances each time step. The accumulator is then shifted right by 20 bits, to get a 12 bit offset into a lookup table. The value in the lookup table at that offset is then put into a buffer for output to the DAC.

Once you have issued the command, Comedi expects you to keep the buffer full of data to output to the acquisition card. This is done by `write()`. Since there may be a delay between the comedi_command() and a subsequent `write()`, you should fill the buffer using `write()` before you call comedi_command(), as is done here.

```
#include <stdio.h>
#include <comedilib.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <getopt.h>
#include <ctype.h>
#include <math.h>
#include "examples.h"

double waveform_frequency = 10.0; /* frequency of the sine wave to output */
double amplitude          = 4000; /* peak-to-peak amplitude, in DAC units (i.e., 0-4095) */
double offset             = 2048; /* offset, in DAC units */

/* This is the size of chunks we deal with when creating and
   outputting data.  This *could* be 1, but that would be
   inefficient */
```

```
#define BUF_LEN 4096

int subdevice;
int external_trigger_number = 0;

sampl_t data[BUF_LEN];

void dds_output(sampl_t *buf,int n);
void dds_init(void);

/* This define determines which waveform to use. */
#define dds_init_function dds_init_sine

void dds_init_sine(void);
void dds_init_pseudocycloid(void);
void dds_init_sawtooth(void);

int comedi_internal_trigger(comedi_t *dev, unsigned int subd, unsigned int trignum)
{
  comedi_insn insn;
  lsampl_t data[1];

  memset(&insn, 0, sizeof(comedi_insn));
  insn.insn = INSN_INTTRIG;
  insn.subdev = subd;
  insn.data = data;
  insn.n = 1;

  data[0] = trignum;

  return comedi_do_insn(dev, &insn);
}


int main(int argc, char *argv[])
{
  comedi_cmd cmd;
  int err;
  int n,m;
  int total=0;
  comedi_t *dev;
  unsigned int chanlist[16];
  unsigned int maxdata;
  comedi_range *rng;
  int ret;
  lsampl_t insn_data = 0;

  parse_options(argc,argv);

  /* Force n_chan to be 1 */
  n_chan = 2;

  if(value){ waveform_frequency = value; }

  dev = comedi_open(filename);
  if(dev == NULL){
    fprintf(stderr, "error opening %s\n", filename);
```

```
  return -1;
}
subdevice = comedi_find_subdevice_by_type(dev,COMEDI_SUBD_AO,0);

maxdata   = comedi_get_maxdata(dev,subdevice,0);
rng       = comedi_get_range(dev,subdevice,0,0);
offset    = (double)comedi_from_phys(0.0,rng,maxdata);
amplitude = (double)comedi_from_phys(1.0,rng,maxdata) - offset;

memset(&cmd,0,sizeof(cmd));
/* fill in the command data structure: */
cmd.subdev         = subdevice;
cmd.flags          = 0;
cmd.start_src      = TRIG_INT;
cmd.start_arg      = 0;
cmd.scan_begin_src = TRIG_TIMER;
cmd.scan_begin_arg = 1e9/freq;
cmd.convert_src    = TRIG_NOW;
cmd.convert_arg    = 0;
cmd.scan_end_src   = TRIG_COUNT;
cmd.scan_end_arg   = n_chan;
cmd.stop_src       = TRIG_NONE;
cmd.stop_arg       = 0;

cmd.chanlist       = chanlist;
cmd.chanlist_len   = n_chan;

chanlist[0] = CR_PACK(channel,range,aref);
chanlist[1] = CR_PACK(channel+1,range,aref);

dds_init();

dds_output(data,BUF_LEN);
dds_output(data,BUF_LEN);

dump_cmd(stdout,&cmd);

if ((err = comedi_command(dev, &cmd)) < 0) {
  comedi_perror("comedi_command");
  exit(1);
}

m=write(comedi_fileno(dev),data,BUF_LEN*sizeof(sampl_t));
if(m<0){
  perror("write");
  exit(1);
}
printf("m=%d\n",m);

ret = comedi_internal_trigger(dev, subdevice, 0);
if(ret<0){
  perror("comedi_internal_trigger\n");
  exit(1);
}

while(1){
  dds_output(data,BUF_LEN);
```

```
    n=BUF_LEN*sizeof(sampl_t);
    while(n>0){
      m=write(comedi_fileno(dev),(void *)data+(BUF_LEN*sizeof(sampl_t)-n),n);
      if(m<0){
        perror("write");
        exit(0);
      }
      printf("m=%d\n",m);
      n-=m;
    }
    total+=BUF_LEN;
  }

  return 0;
}

#define WAVEFORM_SHIFT 16
#define WAVEFORM_LEN (1<<WAVEFORM_SHIFT)
#define WAVEFORM_MASK (WAVEFORM_LEN-1)

sampl_t waveform[WAVEFORM_LEN];

unsigned int acc;
unsigned int adder;

void dds_init(void)
{
  adder=waveform_frequency/freq*(1<<16)*(1<<WAVEFORM_SHIFT);

  dds_init_function();
}

void dds_output(sampl_t *buf,int n)
{
  int i;
  sampl_t *p=buf;


  for(i=0;i<n;i++){
  *p=waveform[(acc>>16)&WAVEFORM_MASK];

  p++;
  acc+=adder;
  }
}


void dds_init_sine(void)
{
  int i;


  for(i=0;i<WAVEFORM_LEN;i++){
  waveform[i]=rint(offset+0.5*amplitude*cos(i*2*M_PI/WAVEFORM_LEN));

  }
}
```

```
/* Yes, I know this is not the proper equation for a cycloid.  Fix it. */
void dds_init_pseudocycloid(void)
{
  int i;
  double t;


  for(i=0;i<WAVEFORM_LEN/2;i++){
  t=2*((double)i)/WAVEFORM_LEN;
  waveform[i]=rint(offset+amplitude*sqrt(1-4*t*t));
  }
  for(i=WAVEFORM_LEN/2;i<WAVEFORM_LEN;i++){
  t=2*(1-((double)i)/WAVEFORM_LEN);
  waveform[i]=rint(offset+amplitude*sqrt(1-t*t));
  }

}

void dds_init_sawtooth(void)
{
  int i;


  for(i=0;i<WAVEFORM_LEN;i++){
  waveform[i]=rint(offset+amplitude*((double)i)/WAVEFORM_LEN);

  }
}
```

4. Acquisition and configuration functions

This Section gives an overview of all Comedi functions with which application programmers can implement their data acquisition. (With “acquisition” we mean all possible kinds of interfacing with the cards: input, output, configuration, streaming, etc.) Section 7 explains the function calls in full detail.

4.1. Functions for single acquisition

The simplest form of using Comedi is to get one single sample to or from an interface card. This sections explains how to do such simple digital and analog acquisitions.

4.1.1. Single digital acquisition

Many boards supported by Comedi have digital input and output channels; i.e., channels that can only produce a `0` or a `1`. Some boards allow the *direction* (input or output) of each channel to be specified independently in software.

Comedi groups digital channels into a *subdevice*, which is a group of digital channels that have the same characteristics. For example, digital output lines will be grouped into a digital output subdevice, bidirectional digital lines will be grouped into a digital I/O subdevice. Thus, there can be multiple digital subdevices on a particular board.

Individual bits on a digital I/O device can be read and written using the functions

```
int comedi_dio_read(device,subdevice,channel,unsigned int *bit);
int comedi_dio_write(device,subdevice,channel,unsigned int bit);
```

The *device* parameter is a pointer to a successfully opened Comedi device. The *subdevice* and *channel* parameters are positive integers that indicate which subdevice and channel is used in the acquisition. The integer *bit* contains the value of the acquired bit.

The direction of bidirectional lines can be configured using the function

```
comedi_dio_config(device,subdevice,channel,unsigned int dir);
```

The parameter *dir* should be either `COMEDI_INPUT` or `COMEDI_OUTPUT`. Many digital I/O subdevices group channels into blocks for configuring direction. Changing one channel in a block changes the entire block.

Multiple channels can be read and written simultaneously using the function

```
comedi_dio_bitfield(device,subdevice,unsigned int write_mask,unsigned int *bits);
```

Each channel is assigned to a bit in the *write_mask* and *bits* bitfield. If a bit in *write_mask* is set, the corresponding bit in **bits* will be written to the corresponding digital output line. Each digital line is then read and placed into **bits*. The value of bits in **bits* corresponding to digital output lines is undefined and device-specific. Channel `0` is the least significant bit in the bitfield; channel `31` is the most significant bit. Channels higher than `31` cannot be accessed using this method.

The digital acquisition functions seem to be very simple, but, behind the implementation screens of the Comedi kernel module, they are executed as special cases of the general instruction command.

4.1.2. Single analog acquisition

Analog Comedi channels can produce data values that are *samples* from continuous analog signals. These samples are integers with a significant content in the range of, typically, `8`, `10`, `12`, or `16` bits.

The

```
int comedi_data_read(comedi_t * device, unsigned int subdevice, unsigned int channel,
                     unsigned int range, unsigned int aref, lsampl_t * data);
```

function reads one such data value from a Comedi channel, and puts it in the user-specified *data* buffer. The

```
int comedi_data_write(comedi_t * device, unsigned int subdevice, unsigned int channel,
                      unsigned int range, unsigned int aref, lsampl_t data);
```

works in the opposite direction. Data values returned by this function are unsigned integers less than, or equal to, the maximum sample value of the channel, which can be determined using the function

```
lsampl_t comedi_get_maxdata(comedi_t * device, unsigned int subdevice, unsigned int channel);
```

Conversion of data values to physical units can be performed by the function

```
double comedi_to_phys(lsampl_t data, comedi_range * range,  lsampl_t maxdata);
```

There are two data structures in these commands that are not fully self-explanatory:

- comedi_t: this data structure contains all information that a user program has to know about an *open* Comedi device. The programmer doesn't have to fill in this data structure manually: it gets filled in by opening the device.
- lsampl_t: this "data structure" represents one single sample. On most architectures, it's nothing more than a 32 bits value. Internally, Comedi does some conversion from raw sample data to "correct" integers. This is called "data munging".

Each single acquisition by, for example, `comedi_data_read()` requires quite some overhead, because all the arguments of the function call are checked. If multiple acquisitions must be done on the same channel, this overhead can be avoided by using a function that can read more than one sample:

```
int comedi_data_read_n(comedi_t *it, unsigned int subdev, unsigned int chan, unsigned int range
    unsigned int aref, lsampl_t *data, unsigned int n)
```

The number of samples, *n*, is limited by the Comedi implementation (to a maximum of 100 samples), because the call is blocking.

The start of the data acquisition can also be delayed by a specified number of nano-seconds:

```
int comedi_data_read_delayed(comedi_t *it, unsigned int subdev, unsigned int chan, unsigned int r
    unsigned int aref, lsampl_t *data, unsigned int nano_sec)
```

All these read and write acquisition functions are implemented on top of the generic instruction command.

4.2. Instructions for multiple acquisitions

The *instruction* is one of the most generic, overloaden and flexible functions in the Comedi API. It is used to execute a multiple of identical acquisitions on the same channel, but also to perform a configuration of a channel. An *instruction list* is a list of instructions, possibly on different channels. Both instructions and instructions lists are executed *synchronously*, i.e., while **blocking** the calling process. This is one of the limitations of instructions; the other one is that they cannot code an acquisition involving timers or external events. These limits are eliminated by the command acquisition primitive.

4.2.1. The instruction data structure

All the information needed to execute an instruction is stored in the comedi_insn data structure:

```
struct comedi_insn_struct{
```

```
  unsigned int insn;      // integer encoding the type of acquisition
                          // (or configuration)
  unsigned int n;         // number of samples
  lsampl_t *data;         // pointer to data buffer
  unsigned int subdev;    // subdevice
  unsigned int chanspec; // encoded channel specification
  unsigned int unused[3];
} comedi_insn;
```

Because of the large flexibility of the instruction function, many types of instruction do not need to fill in all fields, or attach different meanings to the same field. But the current implementation of Comedi requires the data field to be at least one byte long.

The insn flag of the instruction data structure determines the type of acquisition executed in the corresponding instruction:

- INSN_READ: the instruction executes a read on an analog channel.
- INSN_WRITE: the instruction executes a write on an analog channel.
- INSN_BITS: indicates that the instruction must read or write values on multiple digital I/O channels.
- INSN_GTOD: the instruction performs a “Get Time Of Day” acquisition.
- INSN_WAIT: the instruction blocks for a specified number of nanoseconds.

4.2.2. Instruction execution

Once an instruction data structure has been filled in, the corresponding instruction is executed as follows:

```
int comedi_do_insn(comedi_t *it, comedi_insn * instruction);
```

Many Comedi instructions are shortcuts that relieve the programmer from explicitly filling in the data structure and calling the comedi_do_insn function.

The

```
int comedi_do_insnlistcomedi_t *it, comedi_insnlist * list)
```

instruction allows to perform a list of instructions in one function call. The number of instructions in the list is limited in the implementation, because instructions are executed *synchronously*, i.e., the call blocks until the whole instruction (list) has finished.

4.3. Instructions for configuration

Section 4.2 explains how instructions are used to do *acquisition* on channels. This section explains how they are used to *configure* a device. There are various sorts of configurations, and the specific information for each different configuration possibility is to be specified via the data buffer of the instruction data structure. (So, the pointer to a lsampl_t is misused as a pointer to an array with board-specific information.)

Using INSN_CONFIG as the insn flag in an instruction data structure indicates that the instruction will *not perform acquisition* on a channel, but will *configure* that channel. For example, the configuration of digital I/O channels is done as follows. The chanspec field in the comedi_insn data structure, contains the channel to be configured. And data[0] contains either COMEDI_INPUT or COMEDI_OUTPUT, depending on the desired direction of the digital I/O lines. On typical devices, multiple channels are grouped together in blocks for determining their direction. And configuring one channel in a block configures the entire block.

Another example of an INSN_CONFIG instruction is the configuration of the TRIG_OTHER event source.

4.4. Instruction for internal triggering

This special instruction has INSN_INTTRIG as the insn flag in its instruction data structure. Its execution causes an internal triggering event. This event can, for example, cause the device driver to start a conversion, or to stop an ongoing acquisition. The exact meaning of the triggering depends on the card and its particular driver.

The data[0] field of the INSN_INTTRIG instruction is reserved for future use, and should be set to “0”.

4.5. Commands for streaming acquisition

The most powerful Comedi acquisition primitive is the *command*. It’s powerful because, with one single command, the programmer launches:

- a possibly infinite *sequence of acquisitions*,
- accompanied with various *callback* functionalities (DMA, interrupts, driver-specific callback functions),
- for *any number of channels*,
- with an *arbitrary order* of channels in each scan (possibly even with repeated channels per scan),
- and with various scan *triggering sources*, external (i.e., hardware pulses) as well as internal (i.e., pulses generated on the DAQ card itself, or generated by a software trigger instruction).

This command functionality exists in the Comedi API, because various data acquisition devices have the capability to perform this kind of complex acquisition, driven by either on-board or off-board timers and triggers.

A command specifies a particular data acquisition sequence, which consists of a number of *scans*, and each scan is comprised of a number of *conversions*, which usually corresponds to a single A/D or D/A conversion. So, for example, a scan could consist of sampling channels 1, 2 and 3 of a particular device, and this scan should be repeated 1000 times, at intervals of 1 millisecond apart.

The command function is complementary to the configuration instruction function: each channel in the command’s chanlist should first be configured by an appropriate instruction.

4.5.1. Executing a command

A commands is executed by the following Comedi function:

```
int comedi_command(comedi_t * device,  comedi_cmd * command);
```

The following sections explain the meaning of the comedi_cmd data structure. Filling in this structure can be quite complicated, and requires good knowledge about the exact functionalities of the DAQ card. So, before launching a command, the application programmer is adviced to check whether this complex command data structure can be successfully parsed. So, the typical sequence for executing a command is to first send the command through comedi_command_test() once or twice. The test will check that the command is valid for the particular device, and often makes some adjustments to the command arguments, which can then be read back by the user to see the actual values used.

A Comedi program can find out on-line what the command capabilities of a specific device are, by means of the comedi_get_cmd_src_mask() function.

4.5.2. The command data structure

The command executes according to the information about the requested acquisition, which is stored in the comedi_cmd data structure:

```
typedef struct comedi_cmd_struct comedi_cmd;

struct comedi_cmd_struct{
  unsigned int subdev;         // which subdevice to sample
  unsigned int flags;          // encode some configuration possibilities
                               // of the command execution; e.g.,
                               // whether a callback routine is to be
                               // called at the end of the command

  unsigned int start_src;      // event to make the acquisition start
  unsigned int start_arg;      // parameters that influence this start

  unsigned int scan_begin_src; // event to make a particular scan start
  unsigned int scan_begin_arg; // parameters that influence this start‘

  unsigned int convert_src;    // event to make a particular conversion start
  unsigned int convert_arg;    // parameters that influence this start

  unsigned int scan_end_src;   // event to make a particular scan terminate
  unsigned int scan_end_arg;   // parameters that influence this termination

  unsigned int stop_src;       // what make the acquisition terminate
  unsigned int stop_arg;       // parameters that influence this termination

  unsigned int *chanlist;      // pointer to list of channels to be sampled
  unsigned int chanlist_len;   // number of channels to be sampled

  sampl_t *data;               // address of buffer
  unsigned int data_len;       // number of samples to acquire
};
```

The start and end of the whole command acquisition sequence, and the start and end of each scan and of each conversion, is triggered by a so-called *event*. More on these in Section 4.5.3.

The `subdev` member of the comedi_cmd structure is the index of the subdevice the command is intended for. The comedi_find_subdevice_by_type() function can be useful in discovering the index of your desired subdevice.

The chanlist member of the comedi_cmd data structure should point to an array whose number of elements is specificed by chanlist_len (this will generally be the same as the scan_end_arg). The chanlist specifies the sequence of channels and gains (and analog references) that should be stepped through for each scan. The elements of the chanlist array should be initialized by "packing" the channel, range and reference information together with the `CR_PACK()` macro.

The data and data_len members can be safely ignored when issueing commands from a user-space program. They only have meaning when a command is sent from a **kernel** module using the `kcomedilib` interface, in which case they specify the buffer where the driver should write/read its data to/from.

The final member of the comedi_cmd structure is the flags field, i.e., bits in a word that can be bitwise-or'd together. The meaning of these bits are explained in a later section.

4.5.3. The command trigger events

A command is a very versatile acquisition instruction, in the sense that it offers lots of possibilities to let different hardware and software sources determine when acquisitions are started, performed, and stopped. More specifically, the command data structure has *five* types of events: start the acquisition, start a scan, start a conversion, stop a scan, and stop the acquisition. Each event can be given its own *source* (the `*_src` members in the comedi_cmd data structure). And each event source can have a corresponding argument (the `*_arg` members of the comedi_cmd data structure) whose meaning depends on the type of source trigger. For example, to specify an external digital line "3" as a source (in general, *any* of the five event sources), you would use `src`=TRIG_EXT and `arg`=3.

The following paragraphs discuss in somewhat more detail the trigger event sources(`*_src`), and the corresponding arguments (`*_arg`).

The start of an acquisition is controlled by the start_src events. The available options are:

- TRIG_NOW: the start_src event occurs start_arg nanoseconds after the comedi_cmd is called. Currently, only start_arg=0 is supported.
- TRIG_FOLLOW: (For an output device.) The start_src event occurs when data is written to the buffer.
- TRIG_EXT: the start event occurs when an external trigger signal occurs; e.g., a rising edge of a digital line. start_arg chooses the particular digital line.
- TRIG_INT: the start event occurs on a Comedi internal signal, which is typically caused by an INSN_INTTRIG instruction.

The start of the beginning of each scan is controlled by the scan_begin events. The available options are:

- TRIG_TIMER: scan_begin events occur periodically. The time between scan_begin events is convert_arg nanoseconds.

- TRIG_FOLLOW: The scan_begin event occurs immediately after a scan_end event occurs.
- TRIG_EXT: the scan_begin event occurs when an external trigger signal occurs; e.g., a rising edge of a digital line. scan_begin_arg chooses the particular digital line.

The scan_begin_arg used here may not be supported exactly by the device, but it will be adjusted to the nearest supported value by comedi_command_test().

The timing between each sample in a scan is controlled by the convert_* fields:

- TRIG_TIMER: the conversion events occur periodically. The time between convert events is convert_arg nanoseconds.
- TRIG_EXT: the conversion events occur when an external trigger signal occurs, e.g., a rising edge of a digital line. convert_arg chooses the particular digital line.
- TRIG_NOW: All conversion events in a scan occur simultaneously.

The *end* of each scan is almost always specified using TRIG_COUNT, with the argument being the same as the number of channels in the chanlist. You could probably find a device that allows something else, but it would be strange.

The end of an acquisition is controlled by stop_src and stop_arg:

- TRIG_COUNT: stop the acquisition after stop_arg scans.
- TRIG_NONE: perform continuous acquisition, until stopped using comedi_cancel().

  Its argument is reserved and should be set to 0. (“Reserved” means that unspecified things could happen if it is set to something else but 0.)

There are a couple of less usual or not yet implemented events:

- TRIG_TIME: cause an event to occur at a particular time.

  (This event source is reserved for future use.)

- TRIG_OTHER: driver specific event trigger.

  This event can be useful as any of the trigger sources. Its exact meaning is driver specific, because it implements a feature that otherwise does not fit into the generic Comedi command interface. Configuration of TRIG_OTHER features are done by INSN_CONFIG instructions.

  The argument is reserved and should be set to 0.

Not all event sources are applicable to all events. Supported trigger sources for specific events depend significantly on your particular device, and even more on the current state of its device driver. The comedi_get_cmd_src_mask() function is useful for determining what trigger sources a subdevice supports.

4.5.4. The command flags

The flags field in the command data structure is used to specify some “behaviour” of the acquisitions in a command. The meaning of the field is as follows:

- TRIG_RT: ask the driver to use a **hard real-time** interrupt handler. This will reduce latency in handling interrupts from your data aquisition hardware. It can be useful if you are sampling at high frequency, or if your hardware has a small onboard data buffer. You must have a real-time kernel (RTAI (http://www.rtai.org) or RTLinux/Free (http://fsmlabs.com/community/)) and must compile Comedi with real-time support, or this flag will do nothing.
- TRIG_WAKE_EOS: where “EOS” stands for “End of Scan”. Some drivers will change their behaviour when this flag is set, trying to transfer data at the end of every scan (instead of, for example, passing data in chunks whenever the board’s hardware data buffer is half full). This flag may degrade a driver’s performance at high frequencies, because the end of a scan is, in general, a much more frequent event than the filling up of the data buffer.
- TRIG_ROUND_NEAREST: round to nearest supported timing period, the default. This flag (as well as the following three), indicates how timing arguments should be rounded if the hardware cannot achieve the exact timing requested.
- TRIG_ROUND_DOWN: round period down.
- TRIG_ROUND_UP: round period up.
- TRIG_ROUND_UP_NEXT: this one doesn’t do anything, and I don’t know what it was intended to do...?
- TRIG_DITHER: enable dithering? Dithering is a software technique to smooth the influence of discretization “noise”.
- TRIG_DEGLITCH: enable deglitching? Another “noise” smoothing technique.
- TRIG_WRITE: write to bidirectional devices. Could be useful, in principle, if someone wrote a driver that supported commands for a digital I/O device that could do either input or output.
- TRIG_BOGUS: do the motions?
- TRIG_CONFIG: perform configuration, not triggering. This is a legacy of the deprecated comedi_trig_struct data structure, and has no function at present.

4.5.5. Anti-aliasing

If you wish to aquire accurate waveforms, it is vital that you use an anti-alias filter. An anti-alias filter is a low-pass filter used to remove all frequencies higher than the Nyquist frequency (half your sampling rate) from your analog input signal before you convert it to digital. If you fail to filter your input signal, any high frequency components in the original analog signal will create artifacts in your recorded digital waveform that cannot be corrected.

For example, suppose you are sampling an analog input channel at a rate of 1000 Hz. If you were to apply a 900 Hz sine wave to the input, you would find that your sampling rate is not high enough to faithfully record the 900 Hz input, since it is above your Nyquist frequency of 500 Hz. Instead, what you will see in your recorded digital waveform is a 100 Hz sine wave! If you don’t use an anti-alias filter, it is impossible to tell whether the 100 Hz sine wave you see in your digital signal was really produced by a 100 Hz input signal, or a 900 Hz signal aliased to 100 Hz, or a 1100 Hz signal, etc.

In practice, the cutoff frequency for the anti-alias filter is usually set 10% to 20% below the Nyquist frequency due to fact that real filters do not have infinitely sharp cutoffs.

4.6. Slowly-varying inputs

Sometimes, your input channels change slowly enough that you are able to average many successive input values to get a more accurate measurement of the actual value. In general, the more samples you average, the better your estimate gets, roughly by a factor of sqrt(number_of_samples). Obviously, there are limitations to this:

- you are ultimately limited by “Spurious Free Dynamic Range”. This SFDR is one of the popular measures to quantify how much noise a signal carries. If you take a Fourier transform of your signal, you will see several “peaks” in the transform: one or more of the fundamental harmonics of the measured signal, and lots of little “peaks” (called “spurs”) caused by noise. The SFDR is then the difference between the amplitude of the fundamental harmonic and of the largest spur (at frequencies below half of the Nyquist frequency of the DAQ sampler!).
- you need to have *some* noise on the input channel, otherwise you will be averaging the same number `N` times. (Of course, this only holds if the noise is large enough to cause at least a one-bit discretization.)
- the more noise you have, the greater your SFDR, but it takes many more samples to compensate for the increased noise.
- if you feel the need to average samples for, for example, two seconds, your signal will need to be *very* slowly-varying, i.e., not varying more than your target uncertainty for the entire two seconds.

As you might have guessed, the Comedi library has functions to help you in your quest to accurately measure slowly varying inputs:

```
  int comedi_sv_init(comedi_sv_t * sv, comedi_t * device, unsigned int subdevice, unsigned int ch
```

This function initializes the comedi_sv_t data structure, used to do the averaging acquisition:

```
struct comedi_sv_struct{
  comedi_t *dev;
  unsigned int subdevice;
  unsigned int chan;

  /* range policy */
  int range;
  int aref;

  /* number of measurements to average (for analog inputs) */
  int n;

  lsampl_t maxdata;
};
```

The actual acquisition is done with:

```
  int comedi_sv_measure(comedi_sv_t * sv, double * data);
```

The number of samples over which the `comedi_sv_measure()` averages is limited by the implementation (currently the limit is 100 samples).

One typical use for this function is the measurement of thermocouple voltages. And the Comedi self-calibration utility also uses these functions. On some hardware, it is possible to tell it to measure an internal stable voltage reference, which is typically going to be very slowly varying; on the kilosecond time scale or more. So, it is reasonable to measure millions of samples, to get a very accurate measurement of the A/D converter output value that corresponds to the voltage reference. Sometimes, however, this is overkill, since there is no need to perform a part-per-million calibration to a standard that is only accurate to a part-per-thousand.

4.7. Experimental functionality

The following subsections document functionality that has not yet matured. Most of this functionality has even not been implemented yet in any single device driver. This information is included here, in order to stimulate discussion about their API, and to encourage pioneering implementations.

4.7.1. Digital input combining machines

(**Status: experimental (i.e., no driver implements this yet)**)

When one or several digital inputs are used to modify an output value, either an accumulator or a single digital line or bit, a bitfield structure is typically used in the Comedi interface. The digital inputs have two properties, “sensitive” inputs and “modifier” inputs. Edge transitions on sensitive inputs cause changes in the output signal, whereas modifier inputs change the effect of edge transitions on sensitive inputs. Note that inputs can be both modifier inputs and sensitive inputs.

For simplification purposes, it is assumed that multiple digital inputs do not change simultaneously.

The combined state of the modifier inputs determine a modifier state. For each combination of modifier state and sensitive input, there is a set of bits that determine the effect on the output value due to positive or negative transitions of the sensitive input. For each transition direction, there are two bits defined as follows:

00: transition is ignored.
01: accumulator is incremented, or output is set.
10: accumulator is decremented, or output is cleared.
11: reserved.
For example, a simple digital follower is specified by the bit pattern 01 10, because it sets the output on positive transitions of the input, and clears the output on negative transitions. A digital inverter is similarily 10 01. These systems have only one sensitive input.

As another example, a simple up counter, which increments on positive transitions of one input, is specified by 01 00. This system has only one sensitive input.

When multiple digital inputs are used, the inputs are divided into two types, inputs which cause changes in the accumulator, and those that only modify the meaning of transitions on other inputs. Modifier inputs do not require bitfields, but there needs to be a bitfield of length 4*(2^(N-1)) for each edge sensitive input, where N is the total number of inputs. Since N is usually 2 or 3, with only one edge sensitive input, the scaling issues are not significant.

4.7.2. Analog filtering configuration

(Status: design (i.e., no driver implements this yet).)

The insn field of the instruction data structure has not been assigned yet.

The chanspec field of the instruction data structure is ignored.

Some devices have the capability to add white noise (dithering) to analog input measurement. This additional noise can then be averaged out, to get a more accurate measurement of the input signal. It should not be assumed that channels can be separately configured. A simple design can use 1 bit to turn this feature on/off.

Some devices have the capability of changing the glitch characteristics of analog output subsytems. The default (off) case should be where the average settling time is lowest. A simple design can use 1 bit to turn this feature on/off.

Some devices have a configurable analog filters as part of the analog input stage. A simple design can use 1 bit to enable/disable the filter. Default is disabled, i.e., the filter being bypassed, or if the choice is between two filters, the filter with the largest bandwidth.

4.7.3. Analog Output Waveform Generation

(Status: design (i.e., no driver implements this yet).)

The insn field of the instruction data structure has not been assigned yet.

The chanspec field of the instruction data structure is ignored.

Some devices have the ability to cyclicly loop through samples kept in an on-board analog output FIFO. This config should allow the user to enable/disable this mode.

This config should allow the user to configure the number of samples to loop through. It may be necessary to configure the channels used.

4.7.4. Extended Triggering

(Status: alpha.)

The insn field of the instruction data structure has not been assigned yet.

The chanspec field of the instruction data structure is ignored.

This section covers common information for all extended triggering configuration, and doesn't describe a particular type of extended trigger.

Extended triggering is used to configure triggering engines that do not fit into commands. In a typical programming sequence, the application will use configuration instructions to configure an extended trigger, and a command, specifying TRIG_OTHER as one of the trigger sources.

Extended trigger configuration should be designed in such a way that the user can probe for valid parameters, similar to how command testing works. An extended trigger configuration instruction should not configure the hardware directly, rather, the configuration should be saved until the subsequent command is issued. This allows more flexibility for future interface changes.

It has not been decided whether the configuration stage should return a token that is then used as the trigger argument in the command. Using tokens is one method to satisfy the problem that extended trigger configurations may have subtle compatiblity issues with other trigger sources/arguments that can only be determined at command test time. Passing all stages of a command test should only be allowed with a properly configured extended trigger.

Extended triggers must use data[1] as flags. The upper 16 bits are reserved and used only for flags that are common to all extended triggers. The lower 16 bits may be defined by the particular type of extended trigger.

Various types of extended triggers must use data[1] to know which event the extended trigger will be assigned to in the command structure. The possible values are an OR'd mask of the following:

- COMEDI_EV_START
- COMEDI_EV_SCAN_BEGIN
- COMEDI_EV_CONVERT
- COMEDI_EV_SCAN_END
- COMEDI_EV_STOP

4.7.5. Analog Triggering

(Status: alpha. The `ni_mio_common.c` driver implements this feature.)

The insn field of the instruction data structure has not been assigned yet.

The chanspec field of the instruction data structure is ignored.

The data field of the instruction data structure is used as follows:

data[1]: trigger and combining machine configuration.
data[2]: analog triggering signal chanspec.
data[3]: primary analog level.
data[4]: secondary analog level.

Analog triggering is described by a digital combining machine that has two sensitive digital inputs. The sensitive digital inputs are generated by configurable analog comparators. The analog comparators generate a digital 1 when the analog triggering signal is greater than the comparator level. The digital inputs are not modifier inputs. Note, however, there is an effective modifier due to the restriction that the primary analog comparator level must be less than the secondary analog comparator level.

If only one analog comparator signal is used, the combining machine for the secondary input should be set to ignored, and the secondary analog level should be set to 0.

The interpretation of the chanspec and voltage levels is device dependent, but should correspond to similar values of the analog input subdevice, if possible.

Notes: Reading range information is not addressed. This makes it difficult to convert comparator voltages to data values.

Possible extensions: A parameter that specifies the necessary time that the set condition has to be true before the trigger is generated. A parameter that specifies the necessary time that the reset condition has to be true before the state machine is reset.

4.7.6. Bitfield Pattern Matching Extended Trigger

(Status: design. No driver implements this feature yet.)

The insn field of the instruction data structure has not been assigned yet.

The chanspec field of the instruction data structure is ignored.

The data field of the instruction data structure is used as follows:

data[1]: trigger flags.
data[2]: mask.
data[3]: pattern.

The pattern matching trigger issues a trigger when all of a specifed set of input lines match a specified pattern. If the device allows, the input lines should correspond to the input lines of a digital input subdevice, however, this will necessarily be device dependent. Each possible digital line that can be matched is assigned a bit in the mask and pattern. A bit set in the mask indicates that the input line must match the corresponding bit in the pattern. A bit cleared in the mask indicates that the input line is ignored.

Notes: This only allows 32 bits in the pattern/mask, which may be too few. Devices may support selecting different sets of lines from which to match a pattern.

Discovery: The number of bits can be discovered by setting the mask to all 1’s. The driver must modify this value and return -EAGAIN.

4.7.7. Counter configuration

(Status: design. No driver implements this feature yet.)

The insn field of the instruction data structure has not been assigned yet.

The chanspec field of the instruction data structure is used to specify which counter to use. (I.e., the counter is a Comedi channel.)

The data field of the instruction data structure is used as follows:

data[1]: trigger configuration.
data[2]: primary input chanspec.
data[3]: primary combining machine configuration.
data[4]: secondary input chanspec.
data[5]: secondary combining machine configuration.
data[6]: latch configuration.

Note that this configuration is only useful if the counting has to be done in *software*. Many cards offer configurable counters in hardware; e.g., general purpose timer cards can be configured to act as pulse generators, frequency counters, timers, encoders, etc.

Counters can be operated either in synchronous mode (using INSN_READ) or asynchronous mode (using commands), similar to analog input subdevices. The input signal for both modes is the accumulator. Commands on counter subdevices are almost always specified using scan_begin_src = TRIG_OTHER, with the counter configuration also serving as the extended configuration for the scan begin source.

Counters are made up of an accumulator and a combining machine that determines when the accumulator should be incremented or decremented based on the values of the input signals. The combining machine optionally determines when the accumulator should be latched and put into a buffer. This feature is used in asynchronous mode.

Note: How to access multiple pieces of data acquired at each event?

4.7.8. One source plus auxiliary counter configuration

(Status: design. No driver implements this feature yet.)

The insn field of the instruction data structure has not been assigned yet.

The chanspec field of the instruction data structure is used to ...

The data field of the instruction data structure is used as follows:

data[1]: is flags, including the flags for the command triggering configuration. If a command is not subsequently issued on the subd

data[2]: determines the mode of operation. The mode of operation is actually a bitfield that encodes what to do for various transitio
data[3], data[4]: determine the primary source for the counter, similar to the _src and the _arg fields used in the command data stru

Notes: How to specify which events cause a latch and push, and what should get latched?

5. Writing a Comedi driver

This Section explains the most important implementations aspects of the Comedi device drivers. It tries to give the interested device driver writer an overview of the different steps required to write a new device driver.

This Section does *not* explain all implementation details of the Comedi software itself: Comedi has once and for all solved lots of boring but indispensable infrastructural things, such as: timers, management of which drivers are active, memory management for drivers and buffers, wrapping of RTOS-specific interfaces, interrupt handler management, general error handling, the `/proc` interface, etc. So, the device driver writers can concentrate on the interesting stuff: implementing their specific interface card's DAQ functionalities.

In order to make a decent Comedi device driver, you must know the answers to the following questions:

- How does the communication between user space and kernel space work?
- What functionality is provided by the generic kernel-space Comedi functions, and what must be provided for each specific new driver?
- How to use DMA and interrupts?
- What are the addresses and meanings of all the card's registers?

  This information is to be found in the so-called "register level manual" of the card. Without it, coding a device driver is close to hopeless. It is also something that Comedi (and hence also this handbook) cannot give any support or information for: board manufacturers all use their own design and nomenclature.

5.1. Communication user space-kernel space

In user space, you interact with the functions implemented in the `/usr/src/comedilib` directory. Most of the device driver core of the Comedilib library is found in `lib` subdirectory.

All user-space Comedi instructions and commands are transmitted to kernel space through a traditional `ioctl` system call. (See `/usr/src/comedilib/lib/ioctl.c`.) The user space information command is *encoded* as a number in the `ioctl` call, and decoded in the kernel space library. There, they are executed by their kernel-space counterparts. This is done in the `/usr/src/comedi/comedi/comedi_fops.c` file: the `comedi_ioctl()` function processes the results of the `ioctl` system call, interprets its contents, and then calls the corresponding kernel space `do_..._ioctl` function(s). For example, a Comedi instruction is further

processed by the `do_insn_ioctl()`function. (Which, in turn, uses `parse_insn()` for further detailed processing.)

The data corresponding to instructions and commands is transmitted with the `copy_from_user()` system call; acquisition data captured by the interface card passes the kernel-user space boundary with the help of a `copy_to_user()` system call.

5.2. Generic functionality

The major include files of the kernel-space part of Comedi are:

- `include/linux/comedidev.h`: the header file for kernel-only structures (device, subdevice, async (i.e., buffer/event/interrupt/callback functionality for asynchronous DAQ in a Comedi command), driver, lrange), variables, inline functions and constants.
- `include/linux/comedi_rt.h`: all the real-time stuff, such as management of ISR in RTAI and RTLinux/Free, and spinlocks for atomic sections.
- `include/linux/comedilib.h`: the header file for the kernel library of Comedi.

From all the relevant Comedi device driver code that is found in the `/usr/src/comedi/comedi` directory (*if* the Comedi source has been installed in its normal `/usr/src/comedi` location), the **generic** functionality is contained in two parts:

- A couple of `C` files contain the **infrastructural support**. From these `C` files, it's especially the `comedi_fops.c` file that implements what makes Comedi into what people want to use it for: a library that has solved 90% of the DAQ device driver efforts, once and for all.
- For **real-time** applications, the subdirectory `kcomedilib` implements an interface in the kernel that is similar to the Comedi interface accessible through the user-space Comedi library.

  There are some differences in what is possible and/or needed in kernel space and in user space, so the functionalities offered in `kcomedilib` are not an exact copy of the user-space library. For example, locking, interrupt handling, real-time execution, callback handling, etc., are only available in kernel space.

  Most drivers don't make use (yet) of these real-time functionalities.

5.2.1. Data structures

This Section explains the generic data structures that a device driver interacts with:

```
typedef struct comedi_lrange_struct    comedi_lrange;
typedef struct comedi_subdevice_struct comedi_subdevice;
typedef struct comedi_device_struct    comedi_device:
typedef struct comedi_async_struct     comedi_async
typedef struct comedi_driver_struct    comedi_driver;
```

They can be found in `/usr/src/comedi/include/linux/comedidev.h`. Most of the fields are filled in by the Comedi infrastructure, but there are still quite a handful that your driver must provide or use. As for the user-level Comedi, each of the hierarchical layers has its own data structures: channel (`comedi_lrange`), subdevice, and device.

Note that these kernel-space data structures have similar names as their user-space equivalents, but they have a different (kernel-side) view on the DAQ problem and a different meaning: they encode the interaction with the *hardware*, not with the *user*.

However, the comedi_insn and comedi_cmd data structures are shared between user space and kernel space: this should come as no surprise, since these data structures contain all information that the user-space program must transfer to the kernel-space driver for each acquisition.

In addition to these data entities that are also known at the user level (device, sub-device, channel), the device driver level provides two more data structures which the application programmer doesn't get in touch with: the data structure comedi_driver that stores the device driver information that is relevant at the operating system level, and the data structure comedi_async that stores the information about all *asynchronous* activities (interrupts, callbacks and events).

5.2.1.1. `comedi_lrange`

The channel information is simple, since it contains only the signal range information:

```
struct comedi_lrange_struct{
  int           length;
  comedi_krange range[GCC_ZERO_LENGTH_ARRAY];
};
```

5.2.1.2. `comedi_subdevice`

The subdevice is the smallest Comedi entity that can be used for "stand-alone" DAQ, so it is no surprise that it is quite big:

```
struct comedi_subdevice_struct{
  int  type;
  int  n_chan;
  int  subdev_flags;
  int  len_chanlist; /* maximum length of channel/gain list */

  void *private;

  comedi_async *async;

  void         *lock;
  void         *busy;
  unsigned int runflags;

  int          io_bits;
```

```
  lsampl_t maxdata;       /* if maxdata==0, use list */
  lsampl_t *maxdata_list; /* list is channel specific */

  unsigned int flags;
  unsigned int *flaglist;

  comedi_lrange *range_table;
  comedi_lrange **range_table_list;

  unsigned int *chanlist; /* driver-owned chanlist (not used) */

  int (*insn_read)(comedi_device *,comedi_subdevice *,comedi_insn *,lsampl_t *);
  int (*insn_write)(comedi_device *,comedi_subdevice *,comedi_insn *,lsampl_t *);
  int (*insn_bits)(comedi_device *,comedi_subdevice *,comedi_insn *,lsampl_t *);
  int (*insn_config)(comedi_device *,comedi_subdevice *,comedi_insn *,lsampl_t *);

  int (*do_cmd)(comedi_device *,comedi_subdevice *);
  int (*do_cmdtest)(comedi_device *,comedi_subdevice *,comedi_cmd *);
  int (*poll)(comedi_device *,comedi_subdevice *);
  int (*cancel)(comedi_device *,comedi_subdevice *);

  int (*buf_change)(comedi_device *,comedi_subdevice *s,unsigned long new_size);
  void (*munge)(comedi_device *, comedi_subdevice *s, void *data, unsigned int num_bytes, unsigne

  unsigned int state;
};
```

The function pointers `(*insn_read)` ... `(*cancel)` . offer (pointers to) the standardized user-visible API that every subdevice should offer; every device driver has to fill in these functions with their board-specific implementations. (Functionality for which Comedi provides generic functions will, by definition, not show up in the device driver data structures.)

The `buf_change()` and `munge()` functions offer functionality that is not visible to the user and for which the device driver writer must provide a board-specific implementation: `buf_change()` is called when a change in the data buffer requires handling; `munge()` transforms different bit-representations of DAQ values, for example from *unsigned* to *2's complement*.

5.2.1.3. `comedi_device`

The last data structure stores the information at the *device* level:

```
struct comedi_device_struct{
  int           use_count;
  comedi_driver *driver;
  void          *private;
  kdev_t        minor;
  char          *board_name;
  const void    *board_ptr;
  int           attached;
  int           rt;
  spinlock_t    spinlock;
  int           in_request_module;

  int               n_subdevices;
```

```
  comedi_subdevice *subdevices;
  int              options[COMEDI_NDEVCONFOPTS];

  /* dumb */
  int iobase;
  int irq;

  comedi_subdevice *read_subdev;
  wait_queue_head_t read_wait;

  comedi_subdevice *write_subdev;
  wait_queue_head_t write_wait;

  struct fasync_struct *async_queue;

  void (*open)(comedi_device *dev);
  void (*close)(comedi_device *dev);
};
```

5.2.1.4. comedi_async

The following data structure contains all relevant information: addresses and sizes of buffers, pointers to the actual data, and the information needed for event handling:

```
struct comedi_async_struct{
  void *prealloc_buf; /* pre-allocated buffer */
  unsigned int prealloc_bufsz; /* buffer size, in bytes */
  unsigned long *buf_page_list; /* physical address of each page */
  unsigned int max_bufsize; /* maximum buffer size, bytes */
  unsigned int mmap_count; /* current number of mmaps of prealloc_buf */

  volatile unsigned int buf_write_count; /* byte count for writer (write completed) */
  volatile unsigned int buf_write_alloc_count; /* byte count for writer (allocated for writing) *
  volatile unsigned int buf_read_count; /* byte count for reader (read completed)*/

  unsigned int buf_write_ptr; /* buffer marker for writer */
  unsigned int buf_read_ptr; /* buffer marker for reader */

  unsigned int cur_chan; /* useless channel marker for interrupt */
  /* number of bytes that have been received for current scan */
  unsigned int scan_progress;
  /* keeps track of where we are in chanlist as for munging */
  unsigned int munge_chan;

  unsigned int events; /* events that have occurred */

  comedi_cmd cmd;

  // callback stuff
  unsigned int cb_mask;
  int (*cb_func)(unsigned int flags,void *);
  void *cb_arg;

  int (*inttrig)(comedi_device *dev,comedi_subdevice *s,unsigned int x);
```

```
};
```

5.2.1.5. comedi_driver

```
struct comedi_driver_struct{
struct comedi_driver_struct *next;

char *driver_name;
struct module *module;
int (*attach)(comedi_device *,comedi_devconfig *);
int (*detach)(comedi_device *);

/* number of elements in board_name and board_id arrays */
unsigned int num_names;
void *board_name;
/* offset in bytes from one board name pointer to the next */
int offset;
};
```

5.2.2. Generic driver support functions

The directory `comedi` contains a large set of support functions. Some of the most important ones are given below.

From `comedi/comedi_fops.c`, functions to handle the hardware events (which also runs the registered callback function), to get data in and out of the software data buffer, and to parse the incoming functional requests:

```
void comedi_event(comedi_device *dev,comedi_subdevice *s,unsigned int mask);

int comedi_buf_put(comedi_async *async, sampl_t x);
int comedi_buf_get(comedi_async *async, sampl_t *x);

static int parse_insn(comedi_device *dev,comedi_insn *insn,lsampl_t *data,void *file);
```

The file `comedi/kcomedilib/kcomedilib_main.c` provides functions to register a callback, to poll an ongoing data acquisition, and to print an error message:

```
int comedi_register_callback(comedi_t *d,unsigned int subdevice, unsigned int mask,int (*cb)(un

int comedi_poll(comedi_t *d, unsigned int subdevice);

void comedi_perror(const char *message);
```

The file `comedi/rt.c` provides interrupt handling for real-time tasks (one interrupt per *device*!):

```
int comedi_request_irq(unsigned irq,void (*handler)(int, void *,struct pt_regs *), unsigned lon
```

```
void comedi_free_irq(unsigned int irq,comedi_device *dev_id)
```

5.3. Board-specific functionality

The `/usr/src/comedi/comedi/drivers` subdirectory contains the **board-specific** device driver code. Each new card must get an entry in this directory. **Or** extend the functionality of an already existing driver file if the new card is quite similar to that implemented in an already existing driver. For example, many of the National Instruments DAQ cards use the same driver files.

To help device driver writers, Comedi provides the “skeleton” of a new device driver, in the `comedi/drivers/skel.c` file. Before starting to write a new driver, make sure you understand this file, and compare it to what you find in the other already available board-specific files in the same directory.

The first thing you notice in `skel.c` is the documentation section: the Comedi documentation is partially generated automatically, from the information that is given in this section. So, please comply with the structure and the keywords provided as Comedi standards.

The second part of the device driver contains board-specific static data structure and defines: addresses of hardware registers; defines and function prototypes for functionality that is only used inside of the device driver for this board; the encoding of the types and number of available channels; PCI information; etc.

Each driver has to register two functions which are called when you load and unload your board’s device driver (typically via a kernel module):

```
mydriver_attach();
mydriver_detach();
```

In the “attach” function, memory is allocated for the necessary data structures, all properties of a device and its subdevices are defined, and filled in in the generic Comedi data structures. As part of this, pointers to the low level instructions being supported by the subdevice have to be set, which define the basic functionality. In somewhat more detail, the `mydriver_attach()` function must:

- check and request the I/O port region, IRQ, DMA, and other hardware resources. It is convenient here if you verify the existence of the hardware and the correctness of the other information given. Sometimes, unfortunately, this cannot be done.
- allocate memory for the private data structures.
- initialize the board registers and possible subdevices (timer, DMA, PCI, hardware FIFO, etc.).
- return 1, indicating success. If there were any errors along the way, you should return the appropriate error number. If an error is returned, the `mydriver_detach()` function is called. The `mydriver_detach()` function should check any resources that may have been allocated and release them as necessary. The Comedi core frees `dev->subdevices` and `dev->private`, so this does not need to be done in `detach`.
- If the driver has the possibility to offer asynchronous data acquisition, you have to code an interrupt service routine, event handling routines, and/or callback routines.

Typically, you will be able to implement most of the above-mentioned functionality by *cut-and-paste* from already existing drivers. The `mydriver_attach()` function needs most of your attention, because it must correctly define and allocate the (private and generic) data structures that are needed for this device. That is, each sub-device and each channel must get appropriate data fields, and an appropriate initialization. The good news, of course, is that Comedi provides the data structures and the defines that fit very well with almost all DAQ functionalities found on interface cards. These can be found in the header files of the `/usr/src/comedi/include/linux/` directory.

Drivers for digital IOs should implement the following functions:

- `insn_bits()`: drivers set this if they have a function that supports reading and writing multiple bits in a digital I/O subdevice at the same time. Most (if not all) of the drivers use this interface instead of insn_read and insn_write for DIO subdevices.
- `insn_config()`: implements INSN_CONFIG instructions. Currently used for configuring the direction of digital I/O lines, although will eventually be used for generic configuration of drivers that is outside the scope of the currently defined Comedi interface.

Finally, the device driver writer must implement the `read` and `write` functions for the analog channels on the card:

- `insn_read()`: acquire the inputs on the board and transfer them to the software buffer of the driver.
- `insn_write()`: transfer data from the software buffer to the card, and execute the appropriate output conversions.

In some drivers, you want to catch interrupts, and/or want to use the INSN_INTTRIG instruction. In this case, you must provide and register these callback functions.

Implementation of all of the above-mentioned functions requires perfect knowledge about the hardware registers and addresses of the interface card. In general, you can find *some* inspiration in the already available device drivers, but don't trust that blind *cut-and-paste* will bring you far...

5.4. Callbacks, events and interrupts

Continuous acquisition is tyically an *asynchronous* activity: the function call that has set the acquisition in motion has returned before the acquisition has finished (or even started). So, not only the acquired data must be sent back to the user's buffer "in the background", but various types of asynchronous *event handling* can be needed during the acquisition:

- The *hardware* can generate some error or warning events.
- Normal functional interrupts are generated by the hardware, e.g., signalling the filling-up of the card's hardware buffer, or the end of an acquisition scan, etc.
- The device driver writer can register a driver-supplied "callback" function, that is called at the end of each hardware interrupt routine.
- Another driver-supplied callback function is executed when the user program launches an INSN_INTTRIG instruction. This event handling is executed *synchronously* with the execution of the triggering instruction.

The interrupt handlers are registered through the functions mentioned before The event handling is done in the existing Comedi drivers in statements such as this one:

```
s->async->events |= COMEDI_CB_EOA | COMEDI_CB_ERROR
```

It fills in the bits corresponding to particular events in the comedi_async data structure. The possible event bits are:

- *`COMEDI_CB_EOA`*: execute the callback at the “End Of-Acquisition”.
- *`COMEDI_CB_EOS`*: execute the callback at the “End-Of-Scan”.
- *`COMEDI_CB_OVERFLOW`*: execute the callback when a buffer overflow has occurred.
- *`COMEDI_CB_ERROR`*: execute the callback at the occurrence of an (undetermined) error.

5.5. Device driver caveats

A few things to strive for when writing a new driver:

- Some DAQ cards consist of different “layers” of hardware, which can each be given their own device driver. Examples are: some of the National Instruments cards, that all share the same *Mite* PCI driver chip; the ubiquitous parallel port, that can be used for simple digital IO acquisitions. If your new card has such a multi-layer design too, please take the effort to provide drivers for each layer separately.
- Your hardware driver should be functional appropriate to the resources allocated. I.e., if the driver is fully functional when configured with an IRQ and DMA, it should still function moderately well with just an IRQ, or still do minor tasks without IRQ or DMA. Does your driver really require an IRQ to do digital I/O? Maybe someone will want to use your driver *just* to do digital I/O and has no interrupts available.
- Drivers are to have absolutely **no** global variables, mainly because the existence of global variables immediately negates any possibility of using the driver for two devices. The pointer `dev->private` should be used to point to a structure containing any additional variables needed by a driver/device combination.
- Drivers should report errors and warnings via the `comedi_error()` function. (This is *not* the same function as the user-space comedi_perror() function.)

5.6. Integrating the driver in the Comedi library

For integrating new drivers in the Comedi’s source tree the following things have to be done:

- Choose a senseful name for the source code file. Let’s assume here that you call it “mydriver.c”
- Put your new driver into “comedi/drivers/mydriver.c”.
- Edit “comedi/Config.in” and add a new “dep_tristate” line (look at the other examples). Invent a senseful name for the driver’s variable. For example:

```
dep_tristate 'MYDRIVER' CONFIG_COMEDI_MYDRIVER $CONFIG_COMEDI
```

- Add a line to “comedi/drivers/Makefile.in”, using your freshly defined variable, i.e., CONFIG_COMEDI_MYDRIVER.
- Now **make distclean**, reconfigure Comedi with a new **make**, rebuild and be happy.

If you want to have your driver included in the Comedi distribution (you *definitely* want to :-)) send it to David Schleef

<ds@schleef.org>

for review and integration.

6. Low-level drivers

6.1. Low-level drivers

6.1.1. 8255.o -- generic 8255 support

Author: ds

Status: works

Manufacturer	Device	Name
standard	8255	8255

```
The classic in digital I/O.  The 8255 appears in Comedi as a single
digital I/O subdevice with 24 channels.  The channel 0 corresponds
to the 8255's port A, bit 0; channel 23 corresponds to port C, bit
7.  Direction configuration is done in blocks, with channels 0-7,
8-15, 16-19, and 20-23 making up the 4 blocks.  The only 8255 mode
supported is mode 0.

You should enable compilation this driver if you plan to use a board
that has an 8255 chip.  For multifunction boards, the main driver will
configure the 8255 subdevice automatically.

This driver also works independently with ISA and PCI cards that
directly map the 8255 registers to I/O ports, including cards with
multiple 8255 chips.  To configure the driver for such a card, the
option list should be a list of the I/O port bases for each of the
8255 chips.  For example,

  comedi_config /dev/comedi0 8255 0x200,0x204,0x208,0x20c

Note that most PCI 8255 boards do NOT work with this driver, and
need a separate driver as a wrapper.  For those that do work, the
I/O port base address can be found in the output of 'lspci -v'.
```

6.1.2. adl_pci9111.o -- Driver for the Adlink PCI-9111HR card.

Author: Emmanuel Pacaud <emmanuel.pacaud@free.fr>

Status: experimental

Manufacturer	Device	Name
ADLink	PCI-9111HR	adl_pci9111

```
  - ai_insn read
  - ao_insn read/write
  - di_insn read
  - do_insn read/write
  - ai_do_cmd mode with the following sources:

    - start_src  TRIG_NOW
    - scan_begin_src  TRIG_FOLLOW  TRIG_TIMER TRIG_EXT
    - convert_src TRIG_TIMER TRIG_EXT
    - scan_end_src TRIG_COUNT
    - stop_src TRIG_COUNT TRIG_NONE

    The scanned channels must be consecutive and start from 0. They must
    all have the same range and aref.

Configuration options:

    [0] - PCI bus number (optional)
    [1] - PCI slot number (optional)

    If bus/slot is not specified, the first available PCI
    device will be used.
```

6.1.3. adl_pci9118.o -- Adlink PCI-9118DG, PCI-9118HG, PCI-9118HR

Author: Michal Dobes <majkl@tesnet.cz>

Status: works

Manufacturer	Device	Name
ADLink	PCI-9118DG	pci9118dg
ADLink	PCI-9118HG	pci9118hg
ADLink	PCI-9118HR	pci9118hr

```
This driver supports AI, AO, DI and DO subdevices.
AI subdevice supports cmd and insn interface,
other subdevices support only insn interface.
For AI:
- If cmd->scan_begin_src=TRIG_EXT then trigger input is TGIN (pin 46).
- If cmd->convert_src=TRIG_EXT then trigger input is EXTTRG (pin 44).
- If cmd->start_src/stop_src=TRIG_EXT then trigger input is TGIN (pin 46).
- It is not neccessary to have cmd.scan_end_arg=cmd.chanlist_len but
  cmd.scan_end_arg modulo cmd.chanlist_len must by 0.
- If return value of cmdtest is 5 then you've bad channel list
  (it isn't possible mixture S.E. and DIFF inputs or bipolar and unipolar
  ranges).

There are some hardware limitations:
a) You cann't use mixture of unipolar/bipoar ranges or differencial/single
   ended inputs.
b) DMA transfers must have the length aligned to two samples (32 bit),
   so there is some problems if cmd->chanlist_len is odd. This driver tries
   bypass this with adding one sample to the end of the every scan and discard
   it on output but this cann't be used if cmd->scan_begin_src=TRIG_FOLLOW
   and is used flag TRIG_WAKE_EOS, then driver switch to interrupt driven mode
   with interrupt after every sample.
c) If isn't used DMA then you can use only mode where
   cmd->scan_begin_src=TRIG_FOLLOW.

Configuration options:
  [0] - PCI bus of device (optional)
  [1] - PCI slot of device (optional)
          If bus/slot is not specified, then first available PCI
          card will be used.
  [2] - 0= standard 8 DIFF/16 SE channels configuration
        n= external multiplexer connected, 1<=n<=256
  [3] - 0=autoselect DMA or EOC interrupts operation
        1=disable DMA mode
        3=disable DMA and INT, only insn interface will work
  [4] - sample&hold signal - card can generate signal for external S&H board
        0=use SSHO (pin 45) signal is generated in onboard hardware S&H logic
        0!=use ADCHN7 (pin 23) signal is generated from driver, number
           say how long delay is requested in ns and sign polarity of the hold
           (in this case external multiplexor can serve only 128 channels)
  [5] - 0=stop measure on all hardware errors
        2|=ignore ADOR - A/D Overrun status
8|=ignore Bover - A/D Burst Mode Overrun status
256|=ignore nFull - A/D FIFO Full status
```

6.1.4. adv_pci1710.o -- Advantech PCI-1710, PCI-1710HG, PCI-1711, PCI-1713, Advantech PCI-1720, PCI-1731

Author: Michal Dobes <majkl@tesnet.cz>

Status: works

Manufacturer	Device	Name
Advantech	PCI-1710	pci1710
Advantech	PCI-1710HG	pci1710hg
Advantech	PCI-1711	pci1711
Advantech	PCI-1713	pci1713
Advantech	PCI-1720	pci1720
Advantech	PCI-1731	pci1731

```
This driver supports AI, AO, DI and DO subdevices.
AI subdevice supports cmd and insn interface,
other subdevices support only insn interface.

The PCI-1710 and PCI-1710HG have the same PCI device ID, so the
driver cannot distinguish between them, as would be normal for a
PCI driver.

Configuration options:
  [0] - PCI bus of device (optional)
  [1] - PCI slot of device (optional)
          If bus/slot is not specified, the first available PCI
          device will be used.
```

6.1.5. amplc_pc236.o -- Driver for Amplicon PC36AT and PCI236 DIO boards

Author: Ian Abbott <abbotti@mev.co.uk>

Status: works

Manufacturer	Device	Name
Amplicon	PC36AT	pc36at
Amplicon	PCI236	pci236

```
Configuration options - PC36AT:
  [0] - I/O port base address
  [1] - IRQ (optional)
```

```
Configuration options - PCI236:
  [0] - PCI bus of device (optional)
  [1] - PCI slot of device (optional)
  If bus/slot is not specified, the first available PCI device will be
  used.

The PC36AT ISA board and PCI236 PCI board have a single 8255 appearing
as subdevice 0.

Subdevice 1 pretends to be a digital input device, but it always returns
0 when read. However, if you run a command with scan_begin_src=TRIG_EXT,
a rising edge on port C bit 7 acts as an external trigger, which can be
used to wake up tasks.  This is like the comedi_parport device, but the
only way to physically disable the interrupt on the PC36AT is to remove
the IRQ jumper.  If no interrupt is connected, then subdevice 1 is
unused.
```

6.1.6. amplc_pc263.o -- Driver for Amplicon PC263 and PCI263 Relay boards

Author: Ian Abbott <abbotti@mev.co.uk>

Status: works

Manufacturer	Device	Name
Amplicon	PC263	pc263
Amplicon	PCI263	pci263

```
Configuration options - PC263:
  [0] - I/O port base address

Configuration options - PCI263:
  [0] - PCI bus of device (optional)
  [1] - PCI slot of device (optional)
  If bus/slot is not specified, the first available PCI device will be
  used.

Each board appears as one subdevice, with 16 digital outputs, each
connected to a reed-relay. Relay contacts are closed when output is 1.
The state of the outputs can be read.
```

6.1.7. amplc_pci230.o -- Driver for Amplicom PCI230 and PCI260 Multifunction I/O boards

Author: Allan Willcox <allanwillcox@ozemail.com.au>

Status: unknown

Manufacturer	Device	Name
Amplicon	PCI230	amplc_pci230
Amplicon	PCI260	amplc_pci230

6.1.8. cb_pcidas.o -- Driver for the ComputerBoards/MeasurementComputing cards of the PCI-DAS series with the AMCC S5933 PCI controller.

Author: Ivan Martinez <ivanmr@altavista.com>, Frank Mori Hess <fmhess@uiuc.edu>

Status: - PCI-DAS1602/16: Analog input is tested, works. Analog output untested. - PCI-DAS1602/16jr: Driver should work, but untested. Please report usage. - PCI-DAS1602/12: Same as above. - PCI-DAS1200, 1200jr: Tested, works. - PCI-DAS1000, 1001, 1002: Should work, but untested. Please report usage.

Manufacturer	Device	Name
Measurement Computing	PCI-DAS1602/16	cb_pcidas
Measurement Computing	PCI-DAS1602/16jr	cb_pcidas
Measurement Computing	PCI-DAS1602/12	cb_pcidas
Measurement Computing	PCI-DAS1200	cb_pcidas
Measurement Computing	PCI-DAS1200jr	cb_pcidas
Measurement Computing	PCI-DAS1000	cb_pcidas
Measurement Computing	PCI-DAS1001	cb_pcidas
Measurement Computing	PCI_DAS1002	cb_pcidas

```
  The boards' autocalibration features are not yet supported.

Configuration options:
  [0] - PCI bus of device (optional)
  [1] - PCI slot of device (optional)
  If bus/slot is not specified, the first available PCI
  device will be used.

For commands, the scanned channels must be consecutive
(i.e. 4-5-6-7, 2-3-4,...), and must all have the same
range and aref.
```

6.1.9. cb_pcidas64.o -- Driver for the ComputerBoards/MeasurementComputing PCI-DAS64xx, 60XX, and 4020 series with the PLX 9080 PCI controller.

Author: Frank Mori Hess <fmhess@users.sourceforge.net>

Status: works, but no streaming analog output yet

Manufacturer	Device	Name
Measurement Computing	PCI-DAS6402/16	cb_pcidas64
Measurement Computing	PCI-DAS6402/12	cb_pcidas64
Measurement Computing	PCI-DAS64/M1/16	cb_pcidas64
Measurement Computing	PCI-DAS64/M2/16	cb_pcidas64
Measurement Computing	PCI-DAS64/M3/16	cb_pcidas64
Measurement Computing	PCI-DAS6402/16/JR	cb_pcidas64
Measurement Computing	PCI-DAS64/M1/16/JR	cb_pcidas64
Measurement Computing	PCI-DAS64/M2/16/JR	cb_pcidas64
Measurement Computing	PCI-DAS64/M3/16/JR	cb_pcidas64
Measurement Computing	PCI-DAS64/M1/14	cb_pcidas64
Measurement Computing	PCI-DAS64/M2/14	cb_pcidas64
Measurement Computing	PCI-DAS64/M3/14	cb_pcidas64
Measurement Computing	PCI-DAS6023E	cb_pcidas64
Measurement Computing	PCI-DAS6025E	cb_pcidas64
Measurement Computing	PCI-DAS6034E	cb_pcidas64
Measurement Computing	PCI-DAS6035E	cb_pcidas64
Measurement Computing	PCI-DAS4020/12	cb_pcidas64

```
Configuration options:
   [0] - PCI bus of device (optional)
   [1] - PCI slot of device (optional)

Feel free to send and success/failure reports to Frank Hess.

Some devices are not identified because the PCI device IDs are not yet
known. If you have such a board, contact Frank Hess and the ID can be
easily added.
```

6.1.10. cb_pcidda.o -- ComputerBoards/MeasurementComputing PCI-DDA series

Author: Ivan Martinez <ivanmr@altavista.com>, Frank Mori Hess <fmhess@users.sourceforge.net>

Status: Supports 08/16, 04/16, 02/16, 08/12, 04/12, and 02/12

Manufacturer	Device	Name
Measurement Computing	PCI-DDA08/12	cb_pcidda
Measurement Computing	PCI-DDA04/12	cb_pcidda
Measurement Computing	PCI-DDA02/12	cb_pcidda
Measurement Computing	PCI-DDA08/16	cb_pcidda
Measurement Computing	PCI-DDA04/16	cb_pcidda
Measurement Computing	PCI-DDA02/16	cb_pcidda

```
Configuration options:
  [0] - PCI bus of device (optional)
  [1] - PCI slot of device (optional)
  If bus/slot is not specified, the first available PCI
  device will be used.

Only simple analog output writing is supported.

So far it has only been tested with:
  - PCI-DDA08/12
Please report sucess/failure with other different cards to
<comedi@comedi.org>.
```

6.1.11. cb_pcimdda.o -- A driver for this relatively new and uniquely designed board

Author: Calin Culianu <calin@ajvar.org>

Status: works

Manufacturer	Device	Name
Computer Boards	PCIM-DDA06-16	pcimdda06-16

```
All features of the PCIM-DDA06-16 board are supported.  This board
has 6 16-bit AO channels, and the usual 8255 DIO setup.  (24 channels,
configurable in banks of 8 and 4, etc.).  This board does not support commands.

The board has a peculiar way of specifying AO gain/range settings -- You have
1 jumper bank on the card, which either makes all 6 AO channels either
5 Volt unipolar, 5V bipolar, 10 Volt unipolar or 10V bipolar.
```

```
Since there is absolutely _no_ way to tell in software how this jumper is set
(well, at least according  to the rather thin spec. from Measurement Computing
 that comes with the board), the driver assumes the jumper is at its factory
default setting of +/-5V.

Also of note is the fact that this board features another jumper, whose
state is also completely invisible to software.  It toggles two possible AO
output modes on the board:

  - Update Mode: Writing to an AO channel instantaneously updates the actual
    signal output by the DAC on the board (this is the factory default).
  - Simultaneous XFER Mode: Writing to an AO channel has no effect until
    you read from any one of the AO channels.  This is useful for loading
    all 6 AO values, and then reading from any one of the AO channels on the
    device to instantly update all 6 AO values in unison.  Useful for some
    control apps, I would assume?  If your jumper is in this setting, then you
    need to issue your comedi_data_write()s to load all the values you want,
    then issue one comedi_data_read() on any channel on the AO subdevice
    to initiate the simultaneous XFER.


Configuration Options:
  Just tell comedi_config that you want to use the cb_pcimdda driver as so:

  comedi_config /dev/comedi0 cb_pcimdda
```

6.1.12. comedi_parport.o -- Standard PC parallel port

Author: ds

Status: works in immediate mode

Manufacturer	Device	Name
standard	parallel port	comedi_parport

```
A cheap and easy way to get a few more digital I/O lines.  Steal
additional parallel ports from old computers or your neighbors'
computers.

Option list:
 0: I/O port base for the parallel port.
 1: IRQ

Parallel Port Lines:

pin     subdev  chan    aka
---     ------  ----    ---
1       2       0       strobe
```

```
2       0       0       data 0
3       0       1       data 1
4       0       2       data 2
5       0       3       data 3
6       0       4       data 4
7       0       5       data 5
8       0       6       data 6
9       0       7       data 7
10      1       3       acknowledge
11      1       4       busy
12      1       2       output
13      1       1       printer selected
14      2       1       auto LF
15      1       0       error
16      2       2       init
17      2       3       select printer
18-25   ground
```

Subdevices 0 is digital I/O, subdevice 1 is digital input, and subdevice 2 is digital output. Unlike other Comedi devices, subdevice 0 defaults to output.

Pins 13 and 14 are inverted once by Comedi and once by the hardware, thus cancelling the effect.

Pin 1 is a strobe, thus acts like one. There’s no way in software to change this, at least on a standard parallel port.

Subdevice 3 pretends to be a digital input subdevice, but it always returns 0 when read. However, if you run a command with scan_begin_src=TRIG_EXT, it uses pin 10 as a external triggering pin, which can be used to wake up tasks.

6.1.13. comedi_rt_timer.o -- Command emulator using real-time tasks

Author: ds, fmhess

Status: works

This driver requires RTAI or RTLinux to work correctly. It doesn’t actually drive hardware directly, but calls other drivers and uses a real-time task to emulate commands for drivers and devices that are incapable of native commands. Thus, you can get accurately timed I/O on any device.

Since the timing is all done in software, sampling jitter is much higher than with a device that has an on-board timer, and maximum sample rate is much lower.

```
Configuration options:
  [0] - minor number of device you wish to emulate commands for
  [1] - subdevice number you wish to emulate commands for
```

6.1.14. comedi_test.o -- generates fake waveforms

Author: Joachim Wuttke <Joachim.Wuttke@icn.siemens.de>, Frank Mori Hess <fmhess@uiuc.edu>, ds

Status: works

```
This driver is mainly for testing purposes, but can also be used to
generate sample waveforms on systems that don't have data acquisition
hardware.

Configuration options:
  [0] - Amplitude in microvolts for fake waveforms (default 1 volt)
  [1] - Period in microseconds for fake waveforms (default 0.1 sec)

Generates a sawtooth wave on channel 0, square wave on channel 1, additional
waveforms could be added to other channels (currently they return flatline
zero volts).
```

6.1.15. contec_pci_dio.o -- Driver for Contec PIO1616L digital io board

Author: Stefano Rivoir <s.rivoir@gts.it>

Status: works

Manufacturer	**Device**	**Name**
Contec	PIO1616L	contec_pci_dio

```
Configuration Options:
  none
```

6.1.16. daqboard2000.o -- IOTech DAQBoard/2000

Author: Anders Blomdell <anders.blomdell@control.lth.se>

Status: works

Manufacturer	Device	Name
IOTech	DAQBoard/2000	daqboard2000

```
Much of the functionality of this driver was determined from reading
the source code for the Windows driver.

The FPGA on the board requires initialization code, which can either
be compiled into the driver or loaded by comedi_config using the -i
option.  The latter is recommended, in order to save a bit of kernel
memory.

Configuration options:
  [0] - pointer to FPGA initialization data
          The pointer and size options are handled automatically
          by comedi_config when you use the -i option.
  [1] - size of FPGA data
```

6.1.17. das08.o -- DAS-08 compatible boards

Author: Warren Jasper, ds, Frank Hess

Status: works

Manufacturer	Device	Name
ComputerBoards	DAS08	das08
ComputerBoards	DAS08-PGM	das08-pgm
ComputerBoards	DAS08-PGH	das08-pgh
ComputerBoards	DAS08-PGL	das08-pgl
ComputerBoards	DAS08-AOH	das08-aoh
ComputerBoards	DAS08-AOL	das08-aol
ComputerBoards	DAS08-AOM	das08-aom
ComputerBoards	DAS08/JR-AO	das08/jr-ao
ComputerBoards	DAS08/JR-16-AO	das08jr-16-ao
ComputerBoards	PCI-DAS08	pci-das08
ComputerBoards	PCM-DAS08	pcm-das08
ComputerBoards	PC104-DAS08	pc104-das08

Manufacturer	Device	Name
ComputerBoards	DAS08/JR/16	das08jr/16

```
This is a rewrite of the das08 and das08jr drivers.

Options (for ISA cards):
        [0] - base io address

Options (for pci-das08):
        [0] - bus  (optional)
        [1] = slot (optional)
Use the name 'pci-das08' for the pci-das08, NOT 'das08'.

Options (for pcm-das08):
        NONE

The das08 driver doesn't support asynchronous commands, since
the cheap das08 hardware doesn't really support them (except for
pcm-das08).  The
comedi_rt_timer driver can be used to emulate commands for this
driver.
```

6.1.18. das16.o -- DAS16 compatible boards

Author: Sam Moore, Warren Jasper, ds, Chris Baugher, Frank Hess, Roman Fietze

Status: works

Manufacturer	Device	Name
Keithley Metrabyte	DAS-16	das-16
Keithley Metrabyte	DAS-16G	das-16g
Keithley Metrabyte	DAS-16F	das-16f
Keithley Metrabyte	DAS-1201	das-1201
Keithley Metrabyte	DAS-1202	das-1202
Keithley Metrabyte	DAS-1401	das-1401
Keithley Metrabyte	DAS-1402	das-1402
Keithley Metrabyte	DAS-1601	das-1601
Keithley Metrabyte	DAS-1602	das-1602
ComputerBoards	PC104-DAS16/JR	pc104-das16jr
ComputerBoards	PC104-DAS16JR/16	pc104-das16jr/16
ComputerBoards	CIO-DAS16JR/16	cio-das16jr/16
ComputerBoards	CIO-DAS16/JR	cio-das16/jr
ComputerBoards	CIO-DAS1401/12	cio-das1401/12

Manufacturer	Device	Name
ComputerBoards	CIO-DAS1402/12	cio-das1402/12
ComputerBoards	CIO-DAS1402/16	cio-das1402/16
ComputerBoards	CIO-DAS1601/12	cio-das1601/12
ComputerBoards	CIO-DAS1602/12	cio-das1602/12
ComputerBoards	CIO-DAS1602/16	cio-das1602/16
ComputerBoards	CIO-DAS16/330	cio-das16/330

```
A rewrite of the das16 and das1600 drivers.

Passing a zero for an option is the same as leaving it unspecified.

Both a dma channel and an irq (or use of 'timer mode', option 8) are required
for timed or externally triggered conversions.
```

6.1.19. das16m1.o -- CIO-DAS16/M1

Author: Frank Mori Hess <fmhess@uiuc.edu>

Status: works

Manufacturer	Device	Name
MeasurementComputing	CIO-DAS16/M1	cio-das16/m1

```
This driver supports a single board - the CIO-DAS16/M1.
As far as I know, there are no other boards that have
the same register layout.  Even the CIO-DAS16/M1/16 is
significantly different.

I was _barely_ able to reach the full 1 MHz capability
of this board, using a hard real-time interrupt
(set the TRIG_RT flag in your comedi_cmd and use
rtlinux or RTAI).  The board can't do dma, so the bottleneck is
pulling the data across the ISA bus.  I timed the interrupt
handler, and it took my computer ~470 microseconds to pull 512
samples from the board.  So at 1 Mhz sampling rate,
expect your CPU to be spending almost all of its
time in the interrupt handler.

This board has some unusual restrictions for its channel/gain list.  If the
list has 2 or more channels in it, then two conditions must be satisfied:
(1) - even/odd channels must appear at even/odd indices in the list
(2) - the list must have an even number of entries.


irq can be omitted, although the cmd interface will not work without it.
```

6.1.20. das1800.o -- Keithley Metrabyte DAS1800 (& compatibles)

Author: Frank Mori Hess <fmhess@uiuc.edu>

Status: works

Manufacturer	Device	Name
Keithley Metrabyte	DAS-1701ST	das-1701st
Keithley Metrabyte	DAS-1701ST-DA	das-1701st-da
Keithley Metrabyte	DAS-1701/AO	das-1701ao
Keithley Metrabyte	DAS-1702ST	das-1702st
Keithley Metrabyte	DAS-1702ST-DA	das-1702st-da
Keithley Metrabyte	DAS-1702HR	das-1702hr
Keithley Metrabyte	DAS-1702HR-DA	das-1702hr-da
Keithley Metrabyte	DAS-1702/AO	das-1702ao
Keithley Metrabyte	DAS-1801ST	das-1801st
Keithley Metrabyte	DAS-1801ST-DA	das-1801st-da
Keithley Metrabyte	DAS-1801HC	das-1801hc
Keithley Metrabyte	DAS-1801AO	das-1801ao
Keithley Metrabyte	DAS-1802ST	das-1802st
Keithley Metrabyte	DAS-1802ST-DA	das-1802st-da
Keithley Metrabyte	DAS-1802HR	das-1802hr
Keithley Metrabyte	DAS-1802HR-DA	das-1802hr-da
Keithley Metrabyte	DAS-1802HC	das-1802hc
Keithley Metrabyte	DAS-1802AO	das-1802ao

```
The waveform analog output on the 'ao' cards is not supported.
If you need it, send me (Frank Hess) an email.

Configuration options:
  [0] - I/O port base address
  [1] - IRQ (optional, required for timed or externally triggered conversions)
  [2] - DMA0 (optional, requires irq)
  [3] - DMA1 (optional, requires irq and dma0)
```

6.1.21. das6402.o -- Keithley Metrabyte DAS6402 (& compatibles)

Author: Oystein Svendsen <svendsen@pvv.org>

Status: bitrotten

Manufacturer	Device	Name
Keithley Metrabyte	DAS6402	das6402

```
This driver has suffered bitrot.
```

6.1.22. das800.o -- Keithley Metrabyte DAS800 (& compatibles)

Author: Frank Mori Hess <fmhess@uiuc.edu>

Status: works, cio-das802/16 untested - email me if you have tested it

Manufacturer	Device	Name
Keithley Metrabyte	DAS-800	das-800
Keithley Metrabyte	DAS-801	das-801
Keithley Metrabyte	DAS-802	das-802
Measurement Computing	CIO-DAS800	cio-das800
Measurement Computing	CIO-DAS801	cio-das801
Measurement Computing	CIO-DAS802	cio-das802
Measurement Computing	CIO-DAS802/16	cio-das802/16

```
Configuration options:
  [0] - I/O port base address
  [1] - IRQ (optional, required for timed or externally triggered conversions)


All entries in the channel/gain list must use the same gain and be
consecutive channels counting upwards in channel number (these are
hardware limitations.)

I've never tested the gain setting stuff since I only have a
DAS-800 board with fixed gain.

The cio-das802/16 does not have a fifo-empty status bit!  Therefore
only fifo-half-full transfers are possible with this card.
```

6.1.23. dt2801.o -- Data Translation DT2801 series and DT01-EZ

Author: ds

Status: works

Manufacturer	Device	Name
Data Translation	DT2801	dt2801
Data Translation	DT2801-A	dt2801
Data Translation	DT2801/5716A	dt2801
Data Translation	DT2805	dt2801
Data Translation	DT2805/5716A	dt2801
Data Translation	DT2808	dt2801
Data Translation	DT2818	dt2801
Data Translation	DT2809	dt2801
Data Translation	DT01-EZ	dt2801

```
This driver can autoprobe the type of board.

Configuration options:
  [0] - I/O port base address
  [1] - unused
  [2] - A/D reference 0=differential, 1=single-ended
  [3] - A/D range
          0 = [-10,10]
  1 = [0,10]
  [4] - D/A 0 range
          0 = [-10,10]
  1 = [-5,5]
  2 = [-2.5,2.5]
  3 = [0,10]
  4 = [0,5]
  [5] - D/A 1 range (same choices)
```

6.1.24. dt2811.o -- Data Translation DT2811

Author: ds

Status: works

Manufacturer	Device	Name
Data Translation	DT2811-PGL	dt2811-pgl
Data Translation	DT2811-PGH	dt2811-pgh

```
Configuration options:
  [0] - I/O port base address
  [1] - IRQ, although this is currently unused
  [2] - A/D reference
          0 = signle-ended
          1 = differential
  2 = pseudo-differential (common reference)
  [3] - A/D range
          0 = [-5,5]
  1 = [-2.5,2.5]
  2 = [0,5]
  [4] - D/A 0 range (same choices)
  [4] - D/A 1 range (same choices)
```

6.1.25. dt2814.o -- Data Translation DT2814

Author: ds

Status: complete

Manufacturer	Device	Name
Data Translation	DT2814	dt2814

```
Configuration options:
  [0] - I/O port base address
  [1] - IRQ

This card has 16 analog inputs multiplexed onto a 12 bit ADC.  There
is a minimally useful onboard clock.  The base frequency for the
clock is selected by jumpers, and the clock divider can be selected
via programmed I/O.  Unfortunately, the clock divider can only be
a power of 10, from 1 to 10^7, of which only 3 or 4 are useful.  In
addition, the clock does not seem to be very accurate.
```

6.1.26. dt2815.o -- Data Translation DT2815

Author: ds

Status: mostly complete, untested

Manufacturer	Device	Name
Data Translation	DT2815	dt2815

```
I'm not sure anyone has ever tested this board.  If you have information
contrary, please update.

Configuration options:
  [0] - I/O port base base address
  [1] - IRQ (unused)
  [2] - Voltage unipolar/bipolar configuration
          0 == unipolar 5V  (0V -- +5V)
    1 == bipolar 5V  (-5V -- +5V)
  [3] - Current offset configuration
          0 == disabled  (0mA -- +32mAV)
          1 == enabled  (+4mA -- +20mAV)
  [4] - Firmware program configuration
          0 == program 1 (see manual table 5-4)
          1 == program 2 (see manual table 5-4)
          2 == program 3 (see manual table 5-4)
          3 == program 4 (see manual table 5-4)
  [5] - Analog output 0 range configuration
          0 == voltage
          1 == current
  [6] - Analog output 1 range configuration (same options)
  [7] - Analog output 2 range configuration (same options)
  [8] - Analog output 3 range configuration (same options)
  [9] - Analog output 4 range configuration (same options)
  [10] - Analog output 5 range configuration (same options)
  [11] - Analog output 6 range configuration (same options)
  [12] - Analog output 7 range configuration (same options)
```

6.1.27. dt2817.o -- Data Translation DT2817

Author: ds

Status: complete

Manufacturer	Device	Name
Data Translation	DT2817	dt2817

```
A very simple digital I/O card.  Four banks of 8 lines, each bank
is configurable for input or output.  One wonders why it takes a
50 page manual to describe this thing.

The driver (which, btw, is much less than 50 pages) has 1 subdevice
with 32 channels, configurable in groups of 8.

Configuration options:
```

```
  [0] - I/O port base base address
```

6.1.28. dt282x.o -- Data Translation DT2821 series (including DT-EZ)

Author: ds

Status: complete

Manufacturer	Device	Name
Data Translation	DT2821	dt2821
Data Translation	DT2823	dt2823
Data Translation	DT2824-PGH	dt2824-pgh
Data Translation	DT2824-PGL	dt2824-pgl
Data Translation	DT2825	dt2825
Data Translation	DT2827	dt2827
Data Translation	DT2828	dt2828
Data Translation	DT21-EZ	dt21-ez
Data Translation	DT23-EZ	dt23-ez
Data Translation	DT24-EZ	dt24-ez
Data Translation	DT24-EZ-PGL	dt24-ez-pgl

```
Configuration options:
  [0] - I/O port base address
  [1] - IRQ
  [2] - DMA 1
  [3] - DMA 2
  [4] - AI jumpered for 0=single ended, 1=differential
  [5] - AI jumpered for 0=straight binary, 1=2's complement
  [6] - AO 0 jumpered for 0=straight binary, 1=2's complement
  [7] - AO 1 jumpered for 0=straight binary, 1=2's complement
  [8] - AI jumpered for 0=[-10,10]V, 1=[0,10], 2=[-5,5], 3=[0,5]
  [9] - AO 0 jumpered for 0=[-10,10]V, 1=[0,10], 2=[-5,5], 3=[0,5],
        4=[-2.5,2.5]
  [10]- A0 1 jumpered for 0=[-10,10]V, 1=[0,10], 2=[-5,5], 3=[0,5],
        4=[-2.5,2.5]
```

6.1.29. dt3000.o -- Data Translation DT3000 series

Author: ds

Status: works

Manufacturer	Device	Name
Data Translation	DT3001	dt3000
Data Translation	DT3001-PGL	dt3000
Data Translation	DT3002	dt3000
Data Translation	DT3003	dt3000
Data Translation	DT3003-PGL	dt3000
Data Translation	DT3004	dt3000
Data Translation	DT3005	dt3000
Data Translation	DT3004-200	dt3000

```
There is code to support AI commands, but it may not work.

AO commands are not supported.
```

6.1.30. fl512.o -- unknown

Author: unknown

Status: unknown

Manufacturer	Device	Name
unknown	FL512	fl512

```
Digital I/O is not supported.

Configuration options:
  [0] - I/O port base address
```

6.1.31. icp_multi.o -- Inova ICP_MULTI

Author: Anne Smorthit <anne.smorthit@sfwte.ch>

Status: unknown

Manufacturer	Device	Name
Inova	ICP_MULTI	icp_multi

6.1.32. ii_pci20kc.o -- Intelligent Instruments PCI-20001C carrier board

Author: Markus Kempf <kempf@matsci.uni-sb.de>

Status: works

Manufacturer	Device	Name
Intelligent Instrumentation	PCI-20001C	ii_pci20kc

```
Supports the PCI-20001 C-2a Carrier board, and could probably support
the other carrier boards with small modifications.  Modules supported


options for PCI-20006M:
  first:   Analog output channel 0 range configuration
             0  bipolar 10  (-10V -- +10V)
             1  unipolar 10  (0V -- +10V)
             2  bipolar 5  (-5V -- 5V)
  second:  Analog output channel 1 range configuration

options for PCI-20341M:
  first:   Analog input gain configuration
             0  1
             1  10
             2  100
             3  200
```

6.1.33. ke_counter.o -- Driver for Kolter Electronic Counter Card

Author: mh

Status: tested

Manufacturer	Device	Name
Intelligent Instrumentation	PCI Counter Card [ke_counter]	Kolter Electronic

```
This driver is a simple driver to read the counter values from
Kolter Electronic PCI Counter Card.
```

6.1.34. me_daq.o -- Driver for the Meilhaus PCI data acquisition cards.

Author: Michael Hillmann <hillmann@syscongroup.de>

Status: experimental

Manufacturer	Device	Name
Meilhaus	ME-2600i	Kolter Electronic
Meilhaus	ME-2000i	me_daq

```
    Analog Output

Configuration options:

    [0] - PCI bus number (optional)
    [1] - PCI slot number (optional)

    If bus/slot is not specified, the first available PCI
    device will be used.
```

6.1.35. mpc8260cpm.o -- MPC8260 CPM module generic digital I/O lines

Author: ds

Status: experimental

Manufacturer	Device	Name
Motorola	MPC8260 CPM	mpc8260cpm

```
This driver is specific to the Motorola MPC8260 processor, allowing
you to access the processor's generic digital I/O lines.
```

```
It is apparently missing some code.
```

6.1.36. multiq3.o -- Quanser Consulting MultiQ-3

Author: Anders Blomdell <anders.blomdell@control.lth.se>

Status: works

Manufacturer	Device	Name
Quanser Consulting	MultiQ-3	multiq3

6.1.37. ni_670x.o -- National Instruments 670x

Author: Bart Joris <bjoris@advalvas.be>

Status: unknown

Manufacturer	Device	Name
National Instruments	PCI-6703	ni_670x
National Instruments	PCI-6704	ni_670x

```
The driver currently does not recognize the 6704, because the PCI
ID is not known.

Commands are not supported.
```

6.1.38. ni_at_a2150.o -- National Instruments AT-A2150

Author: Frank Mori Hess

Status: works

Manufacturer	Device	Name
National Instruments	AT-A2150C	at_a2150c
National Instruments	AT-2150S	at_a2150s

```
If you want to ac couple the board's inputs, use AREF_OTHER.

Configuration options:
  [0] - I/O port base address
  [1] - IRQ (optional, required for timed conversions)
  [2] - DMA (optional, required for timed conversions)
```

6.1.39. ni_at_ao.o -- National Instruments AT-AO-6/10

Author: ds

Status: untested

Manufacturer	Device	Name
National Instruments	AT-AO-6	at-ao-6
National Instruments	AT-AO-10	at-ao-10

```
This driver has not been tested, but should work.
```

6.1.40. ni_atmio.o -- National Instruments AT-MIO-E series

Author: ds

Status: works

Manufacturer	Device	Name
National Instruments	AT-MIO-16E-1	ni_atmio
National Instruments	AT-MIO-16E-2	ni_atmio
National Instruments	AT-MIO-16E-10	ni_atmio
National Instruments	AT-MIO-16DE-10	ni_atmio

Manufacturer	Device	Name
National Instruments	AT-MIO-64E-3	ni_atmio
National Instruments	AT-MIO-16XE-50	ni_atmio
National Instruments	AT-MIO-16XE-10	ni_atmio
National Instruments	AT-AI-16XE-10	ni_atmio

```
The isapnptools package is required to use this board.  Use isapnp to
configure the I/O base for the board, and then pass the same value as
a parameter in comedi_config.  A sample isapnp.conf file is included
in the etc/ directory of Comedilib.

Comedilib includes a utility to autocalibrate these boards.  The
boards seem to boot into a state where the all calibration DACs
are at one extreme of their range, thus the default calibration
is terrible.  Calibration at boot is strongly encouraged.

To use the extended digital I/O on some of the boards, enable the
8255 driver when configuring the Comedi source tree.

External triggering is supported for some events.  The channel index
(scan_begin_arg, etc.) maps to PFI0 - PFI9.

Some of the more esoteric triggering possibilities of these boards
are not supported.
```

6.1.41. ni_atmio16d.o -- National Instruments AT-MIO-16D

Author: Chris R. Baugher <baugher@enteract.com>

Status: unknown

Manufacturer	Device	Name
National Instruments	AT-MIO-16	atmio16
National Instruments	AT-MIO-16D	atmio16d

6.1.42. ni_daq_dio24.o -- National Instruments PCMCIA DAQ-Card DIO-24

Author: Daniel Vecino Castel <dvecino@able.es>

Status: ?

Manufacturer	Device	Name
National Instruments	National Instruments PCMCIA DAQ-Card DIO-24	atmio16d

6.1.43. ni_labpc.o -- National Instruments Lab-PC (& compatibles)

Author: Frank Mori Hess <fmhess@users.sourceforge.net>

Status: works

Manufacturer	Device	Name
National Instruments	DAQCard-1200	daqcard-1200
National Instruments	Lab-PC-1200	labpc-1200
National Instruments	Lab-PC-1200AI	labpc-1200ai
National Instruments	Lab-PC+	lab-pc+
National Instruments	PCI-1200	pci-1200

```
Tested with lab-pc-1200.  For the older Lab-PC+, not all input ranges
and analog references will work, the available ranges/arefs will
depend on how you have configured the jumpers on your board
(see your owner's manual).

Configuration options - ISA boards:
  [0] - I/O port base address
  [1] - IRQ (optional, required for timed or externally triggered conversions)
  [2] - DMA channel (optional)

Configuration options - PCI boards:
  [0] - bus (optional)
  [1] - slot (optional)

Configuration options - PCMCIA boards:
  none

Lab-pc+ has quirky chanlist when scanning multiple channels.  Scan
sequence must start at highest channel, then decrement down to
channel 0.  1200 series cards can scan down like lab-pc+ or scan
up from channel zero.
```

6.1.44. ni_mio_cs.o -- National Instruments DAQCard E series

Author: ds

Status: works

Manufacturer	Device	Name
National Instruments	DAQCard-AI-16XE-50	ni_mio_cs
National Instruments	DAQCard-AI-16E-4	ni_mio_cs
National Instruments	DAQCard-6062E	ni_mio_cs
National Instruments	DAQCard-6024E	ni_mio_cs

```
See the notes in the ni_atmio.o driver.
```

6.1.45. ni_pcidio.o -- National Instruments PCI-DIO32HS, PCI-DIO96, PCI-6533, PCI-6503

Author: ds

Status: works

Manufacturer	Device	Name
National Instruments	PCI-DIO-32HS	ni_pcidio
National Instruments	PXI-6533	ni_pcidio
National Instruments	PCI-DIO-96	ni_pcidio
National Instruments	PCI-DIO-96B	ni_pcidio
National Instruments	PXI-6508	ni_pcidio
National Instruments	PCI-6503	ni_pcidio
National Instruments	PCI-6503B	ni_pcidio
National Instruments	PCI-6503X	ni_pcidio
National Instruments	PXI-6503	ni_pcidio
National Instruments	PCI-6534	ni_pcidio
National Instruments	PCI-6533	ni_pcidio

```
The DIO-96 appears as four 8255 subdevices.  See the 8255
driver notes for details.

The DIO32HS board appears as one subdevice, with 32 channels.
Each channel is individually I/O configurable.  The channel order
is 0=A0, 1=A1, 2=A2, ... 8=B0, 16=C0, 24=D0.  The driver only
supports simple digital I/O; no handshaking is supported.
```

```
DMA mostly works for the PCI-DIO32HS, but only in timed input mode.

This driver could be easily modified to support AT-MIO32HS and
AT-MIO96.
```

6.1.46. ni_pcimio.o -- National Instruments PCI-MIO-E series (all boards)

Author: ds

Status: works

Manufacturer	**Device**	**Name**
National Instruments	PCI-MIO-16XE-50	ni_pcimio
National Instruments	PCI-MIO-16XE-10	ni_pcimio
National Instruments	PXI-6030E	ni_pcimio
National Instruments	PCI-MIO-16E-1	ni_pcimio
National Instruments	PCI-MIO-16E-4	ni_pcimio
National Instruments	PCI-6040E	ni_pcimio
National Instruments	PXI-6040E	ni_pcimio
National Instruments	PCI-6031E	ni_pcimio
National Instruments	PCI-6032E	ni_pcimio
National Instruments	PCI-6033E	ni_pcimio
National Instruments	PCI-6071E	ni_pcimio
National Instruments	PCI-6023E	ni_pcimio
National Instruments	PCI-6024E	ni_pcimio
National Instruments	PCI-6025E	ni_pcimio
National Instruments	PXI-6025E	ni_pcimio
National Instruments	PCI-6034E	ni_pcimio
National Instruments	PCI-6035E	ni_pcimio
National Instruments	PCI-6052E	ni_pcimio
National Instruments	PCI-6110	ni_pcimio
National Instruments	PCI-6111	ni_pcimio
National Instruments	PCI-6711	ni_pcimio
National Instruments	PCI-6713	ni_pcimio
National Instruments	PXI-6071E	ni_pcimio
National Instruments	PXI-6070E	ni_pcimio
National Instruments	PXI-6052E	ni_pcimio
National Instruments	PCI-6036E	ni_pcimio
National Instruments	PCI-6731	ni_pcimio

Manufacturer	Device	Name
National Instruments	PCI-6733	ni_pcimio

```
These boards are almost identical to the AT-MIO E series, except that
they use the PCI bus instead of ISA (i.e., AT).  See the notes for
the ni_atmio.o driver for additional information about these boards.

Autocalibration is supported on many of the devices, using the
calibration utility in Comedilib.

By default, the driver uses DMA to transfer analog input data to
memory.  When DMA is enabled, not all triggering features are
supported.

Streaming analog output is not supported on PCI-671x and PCI-673x.

PCI IDs are not known for PCI-6731 and PCI-6733.  Digital I/O may not
work on 673x.

Information (number of channels, bits, etc.) for some devices may be
incorrect.  Please check this and submit a bug if there are problems
for your device.
```

6.1.47. pcl711.o -- Advantech PCL-711 and 711b, ADLink ACL-8112

Author: ds, Janne Jalkanen <jalkanen@cs.hut.fi>, Eric Bunn <ebu@cs.hut.fi>

Status: mostly complete

Manufacturer	Device	Name
Advantech	PCL-711	pcl711
Advantech	PCL-711B	pcl711b
AdLink	ACL-8112HG	acl8112hg
AdLink	ACL-8112DG	acl8112dg

```
Since these boards do not have DMA or FIFOs, only immediate mode is
supported.
```

6.1.48. pcl724.o -- Advantech PCL-724, PCL-722, PCL-731 ADLink ACL-7122, ACL-7124, PET-48DIO

Author: Michal Dobes <majkl@tesnet.cz>

Status: untested

Manufacturer	Device	Name
Advantech	PCL-724	pcl724
Advantech	PCL-722	pcl722
Advantech	PCL-731	pcl731
ADLink	ACL-7122	acl7122
ADLink	ACL-7124	acl7124
ADLink	PET-48DIO	pet48dio

```
This is driver for digital I/O boards PCL-722/724/731 with 144/24/48 DIO
and for digital I/O boards ACL-7122/7124/PET-48DIO with 144/24/48 DIO.
It need 8255.o for operations and only immediate mode is supported.
See the source for configuration details.
```

6.1.49. pcl725.o -- Advantech PCL-725 (& compatibles)

Author: ds

Status: unknown

Manufacturer	Device	Name
Advantech	PCL-725	pcl725

6.1.50. pcl726.o -- Advantech PCL-726 & compatibles

Author: ds

Status: untested

Manufacturer	Device	Name
Advantech	PCL-726	pcl726
Advantech	PCL-727	pcl727
Advantech	PCL-728	pcl728
ADLink	ACL-6126	acl6126
ADLink	ACL-6128	acl6128

```
Interrupts are not supported.

    Options for PCL-726:
     [0] - IO Base
     [2]...[7] - D/A output range for channel 1-6:
               0: 0-5V, 1: 0-10V, 2: +/-5V, 3: +/-10V,
       4: 4-20mA, 5: unknown (external reference)

    Options for PCL-727:
     [0] - IO Base
     [2]...[13] - D/A output range for channel 1-12:
               0: 0-5V, 1: 0-10V, 2: +/-5V,
       3: 4-20mA

    Options for PCL-728 and ACL-6128:
     [0] - IO Base
     [2], [3] - D/A output range for channel 1 and 2:
               0: 0-5V, 1: 0-10V, 2: +/-5V, 3: +/-10V,
       4: 4-20mA, 5: 0-20mA

    Options for ACL-6126:
     [0] - IO Base
     [1] - IRQ (0=disable, 3, 5, 6, 7, 9, 10, 11, 12, 15) (currently ignored)
     [2]...[7] - D/A output range for channel 1-6:
               0: 0-5V, 1: 0-10V, 2: +/-5V, 3: +/-10V,
       4: 4-20mA
```

6.1.51. pcl812.o -- Advantech PCL-812/PG, PCL-813/B, ADLink ACL-8112DG/HG/PG, ACL-8113, ACL-8216, ICP DAS A-821PGH/PGL/PGL-NDA, A-822PGH/PGL, A-823PGH/PGL, A-826PG, ICP DAS ISO-813

Author: Michal Dobes <majkl@tesnet.cz>

Status: works (I hope. My board fire up under my hands and I cann’t test all features.)

Manufacturer	Device	Name
Advantech	PCL-812	pcl812
Advantech	PCL-812PG	pcl812pg

Manufacturer	Device	Name
Advantech	PCL-813	pcl813
Advantech	PCL-813B	pcl813b
ADLink	ACL-8112DG	acl8112dg
ADLink	ACL-8112HG	acl8112hg
ADLink	ACL-8113	acl-8113
ADLink	ACL-8216	acl8216
ICP	ISO-813	iso813
ICP	A-821PGH	a821pgh
ICP	A-821PGL	a821pgl
ICP	A-821PGL-NDA	a821pclnda
ICP	A-822PGH	a822pgh
ICP	A-822PGL	a822pgl
ICP	A-823PGH	a823pgh
ICP	A-823PGL	a823pgl
ICP	A-826PG	a826pg

```
This driver supports insn and cmd interfaces. Some boards support only insn
becouse their hardware don't allow more (PCL-813/B, ACL-8113, ISO-813).
Data transfer over DMA is supported only when you measure only one
channel, this is too hardware limitation of these boards.
See the head of the source file pcl812.c for configuration options.
```

6.1.52. pcl816.o -- Advantech PCL-816 cards, PCL-814

Author: Juan Grigera <juan@grigera.com.ar>

Status: works

Manufacturer	Device	Name
Advantech	PCL-816	pcl816
Advantech	PCL-814B	pcl814b

```
PCL 816 and 814B have 16 SE/DIFF ADCs, 16 DACs, 16 DI and 16 DO.
Differences are at resolution (16 vs 12 bits).

The driver support AI command mode, other subdevices not written.

Analog output and digital input and output are not supported.

Configuration Options:
  [0] - IO Base
  [1] - IRQ (0=disable, 2, 3, 4, 5, 6, 7)
```

```
  [2] - DMA (0=disable, 1, 3)
  [3] - 0, 10=10MHz clock for 8254
            1= 1MHz clock for 8254
```

6.1.53. pcl818.o -- Advantech PCL-818 cards, PCL-718

Author: Michal Dobes <majkl@tesnet.cz>

Status: works

Manufacturer	Device	Name
Advantech	PCL-818L	pcl818l
Advantech	PCL-818H	pcl818h
Advantech	PCL-818HD	pcl818hd
Advantech	PCL-818HG	pcl818hg
Advantech	PCL-818	pcl818
Advantech	PCL-718	pcl718

```
All cards have 16 SE/8 DIFF ADCs, one or two DACs, 16 DI and 16 DO.
Differences are only at maximal sample speed, range list and FIFO
support.
The driver support AI mode 0, 1, 3 other subdevices (AO, DI, DO) support
only mode 0. If DMA/FIFO/INT are disabled then AI support only mode 0.
PCL-818HD and PCL-818HG support 1kword FIFO. Driver support this FIFO
but this code is untested.
A word or two about DMA. Driver support DMA operations at two ways:
1) DMA uses two buffers and after one is filled then is generated
   INT and DMA restart with second buffer. With this mode I'm unable run
   more that 80Ksamples/secs without data dropouts on K6/233.
2) DMA uses one buffer and run in autoinit mode and the data are
   from DMA buffer moved on the fly with 2kHz interrupts from RTC.
   This mode is used if the interrupt 8 is available for allocation.
   If not, then first DMA mode is used. With this I can run at
   full speed one card (100ksamples/secs) or two cards with
   60ksamples/secs each (more is problem on account of ISA limitations).
   To use this mode you must have compiled  kernel with disabled
   "Enhanced Real Time Clock Support".
   Maybe you can have problems if you use xntpd or similar.
   If you've data dropouts with DMA mode 2 then:
    a) disable IDE DMA
    b) switch text mode console to fb.

   Options for PCL-818L:
    [0] - IO Base
    [1] - IRQ (0=disable, 2, 3, 4, 5, 6, 7)
    [2] - DMA (0=disable, 1, 3)
```

```
  [3] - 0, 10=10MHz clock for 8254
            1= 1MHz clock for 8254
  [4] - 0,  5=A/D input  -5V.. +5V
        1, 10=A/D input -10V..+10V
  [5] - 0,  5=D/A output 0-5V  (internal reference -5V)
        1, 10=D/A output 0-10V (internal reference -10V)
2    =D/A output unknow (external reference)

 Options for PCL-818, PCL-818H:
  [0] - IO Base
  [1] - IRQ (0=disable, 2, 3, 4, 5, 6, 7)
  [2] - DMA (0=disable, 1, 3)
  [3] - 0, 10=10MHz clock for 8254
            1= 1MHz clock for 8254
  [4] - 0,  5=D/A output 0-5V  (internal reference -5V)
        1, 10=D/A output 0-10V (internal reference -10V)
2    =D/A output unknow (external reference)

 Options for PCL-818HD, PCL-818HG:
  [0] - IO Base
  [1] - IRQ (0=disable, 2, 3, 4, 5, 6, 7)
  [2] - DMA/FIFO  (-1=use FIFO, 0=disable both FIFO and DMA,
                   1=use DMA ch 1, 3=use DMA ch 3)
  [3] - 0, 10=10MHz clock for 8254
            1= 1MHz clock for 8254
  [4] - 0,  5=D/A output 0-5V  (internal reference -5V)
        1, 10=D/A output 0-10V (internal reference -10V)
    2    =D/A output unknow (external reference)

 Options for PCL-718:
  [0] - IO Base
  [1] - IRQ (0=disable, 2, 3, 4, 5, 6, 7)
  [2] - DMA (0=disable, 1, 3)
  [3] - 0, 10=10MHz clock for 8254
            1= 1MHz clock for 8254
  [4] -     0=A/D Range is +/-10V
    1=              +/-5V
    2=              +/-2.5V
    3=              +/-1V
    4=              +/-0.5V
    5=        user defined bipolar
    6=      0-10V
    7=      0-5V
      8=      0-2V
    9=      0-1V
   10=      user defined unipolar
  [5] - 0,  5=D/A outputs 0-5V  (internal reference -5V)
        1, 10=D/A outputs 0-10V (internal reference -10V)
    2=D/A outputs unknow (external reference)
  [6] - 0, 60=max  60kHz A/D sampling
        1,100=max 100kHz A/D sampling (PCL-718 with Option 001 installed)
```

6.1.54. pcm3730.o -- PCM3730

Author: Blaine Lee

Status: unknown

Manufacturer	Device	Name
Advantech	PCM-3730	pcm3730

```
Configuration options:
  [0] - I/O port base
```

6.1.55. pcmad.o -- Winsystems PCM-A/D12, PCM-A/D16

Author: ds

Status: untested

Manufacturer	Device	Name
Winsystems	PCM-A/D12	pcmad12
Winsystems	PCM-A/D16	pcmad16

```
This driver was written on a bet that I couldn't write a driver
in less than 2 hours.  I won the bet, but never got paid.  =(

Configuration options:
  [0] - I/O port base
  [1] - unused
  [2] - Analog input reference
          0 = single ended
          1 = differential
  [3] - Analog input encoding (must match jumpers)
          0 = straight binary
          1 = two's complement
```

6.1.56. poc.o -- Generic driver for very simple devices Device names: dac02

Author: ds

Status: unknown

Manufacturer	Device	Name
Keithley Metrabyte	DAC-02	dac02
Advantech	PCL-733	pcl733
Advantech	PCL-734	pcl734

```
This driver is indended to support very simple ISA-based devices,

Configuration options:
  [0] - I/O port base
```

6.1.57. quatech_daqp_cs.o -- Quatech DAQP PCMCIA data capture cards

Author: Brent Baccala <baccala@freesoft.org>

Status: unkown

Manufacturer	Device	Name
Quatech	DAQP-208	daqp
Quatech	DAQP-308	daqp

6.1.58. rtd520.o -- Real Time Devices PCI4520/DM7520

Author: Dan Christian

Status: Works. Only tested on DM7520-8. Not SMP safe.

Manufacturer	Device	Name
Real Time Devices	DM7520HR-1	DM7520
Real Time Devices	DM7520HR-8	DM7520-8
Real Time Devices	PCI4520	PCI4520
Real Time Devices	PCI4520-8	PCI4520-8

```
Configuration options:
  [0] - PCI bus of device (optional)
          If bus/slot is not specified, the first available PCI
```

```
          device will be used.
  [1] - PCI slot of device (optional)
```

6.1.59. rti800.o -- Analog Devices RTI-800/815

Author: ds

Status: unknown

Manufacturer	Device	Name
Analog Devices	RTI-800	rti800
Analog Devices	RTI-815	rti815

```
Configuration options:
  [0] - I/O port base address
  [1] - IRQ
  [2] - A/D reference
          0 = differential
          1 = pseudodifferential (common)
          2 = single-ended
  [3] - A/D range
          0 = [-10,10]
          1 = [-5,5]
          2 = [0,10]
  [4] - A/D encoding
          0 = two’s complement
          1 = straight binary
  [5] - DAC 0 range
          0 = [-10,10]
          1 = [0,10]
  [5] - DAC 0 encoding
          0 = two’s complement
          1 = straight binary
  [6] - DAC 1 range (same as DAC 0)
  [7] - DAC 1 encoding (same as DAC 0)
```

6.1.60. rti802.o -- Analog Devices RTI-802

Author: Anders Blomdell <anders.blomdell@control.lth.se>

Status: works

Manufacturer	Device	Name
Analog Devices	RTI-802	rti802

```
Configuration Options:
    [0] - i/o base
    [1] - unused
    [2] - dac#0  0=two’s comp, 1=straight
    [3] - dac#0  0=bipolar, 1=unipolar
    [4] - dac#1 ...
    ...
    [17] - dac#7 ...
```

6.1.61. serial2002.o -- Driver for serial connected hardware

Author: Anders Blomdell

Status: in development

6.1.62. skel.o -- Skeleton driver, an example for driver writers

Author: ds

Status: works

```
This driver is a documented example on how Comedi drivers are
written.

Configuration Options:
  none
```

6.1.63. ssv_dnp.o -- SSV Embedded Systems DIL/Net-PC

Author: Robert Schwebel <robert@schwebel.de>

Status: unknown

Manufacturer	Device	Name
SSV Embedded Systems	DIL/Net-PC 1486	dnp-1486

7. Comedi Reference

Reference for constants and macros, data types and structures, and functions.

7.1. Headerfiles: `comedi.h` and `comedilib.h`

All application programs must include the header file `comedilib.h`. (This file itself includes `comedi.h`.) They contain the full interface of Comedi: defines, function prototypes, data structures.

The following Sections give more details.

7.2. Constants and Macros

7.2.1. CR_PACK

CR_PACK is used to initialize the elements of the *`chanlist`* array in the comedi_cmd data structure, and the *`chanspec`* member of the comedi_insn structure.

```
#define CR_PACK(chan,rng,aref)      ( (((aref)&0x3)<<24) | (((rng)&0xff)<<16) | (chan) )
```

The *`chan`* argument is the channel you wish to use, with the channel numbering starting at zero.

The range *`rng`* is an index, starting at zero, whose meaning is device dependent. The comedi_get_n_ranges() and comedi_get_range() functions are useful in discovering information about the available ranges.

The *`aref`* argument indicates what reference you want the device to use. It can be any of the following:

AREF_GROUND

is for inputs/outputs referenced to ground.

AREF_COMMON

is for a “common” reference (the low inputs of all the channels are tied together, but are isolated from ground).

AREF_DIFF

is for differential inputs/outputs.

AREF_OTHER

is for any reference that does not fit into the above categories.

Particular drivers may or may not use the AREF flags. If they are not supported, they are silently ignored.

7.2.2. RANGE_LENGTH (deprecated)

Rangetype values are library-internal tokens that represent an array of range information structures. These numbers are primarily used for communication between the kernel and library.

The RANGE_LENGTH() macro returns the length of the array that is specified by the rangetype token.

The RANGE_LENGTH() macro is deprecated, and should not be used in new applications. It is scheduled to be removed from the header file at version 1.0. Binary compatibility may be broken for version 1.1.

7.3. Data Types and Structures

This Section explains the data structures that users of the Comedi API are confronted with:

```
typedef struct subdevice_struct       subdevice_struct:
typedef struct comedi_devinfo_struct  comedi_devinfo;
typedef struct comedi_t_struct        comedi_t;
typedef struct sampl_t_struct         sampl_t;
typedef struct lsampl_t_struct        lsampl_t;
typedef struct comedi_sv_t_struct     comedi_sv_t;
typedef struct comedi_cmd_struct      comedi_cmd;
typedef struct comedi_insn_struct     comedi_insn;
typedef struct comedi_range_struct    comedi_range;
typedef struct comedi_krange_struct   comedi_krange;
typedef struct comedi_insnlist_struct comedi_insnlist;
```

The data structures used in the implementation of the Comedi drivers are treated elsewhere.

7.3.1. subdevice_struct

The data type *subdevice_struct* is used to store information about a subdevice. This structure is usually filled in automatically when the driver is loaded (“attached”), so programmers need not access this data structure directly.

```
typedef struct subdevice_struct subdevice;

struct subdevice_struct{
  unsigned int type;
  unsigned int n_chan;
  unsigned int subd_flags;
  unsigned int timer_type;
  unsigned int len_chanlist;
  lsampl_t maxdata;
  unsigned int flags;
  unsigned int range_type;

  lsampl_t *maxdata_list;
  unsigned int *range_type_list;
  unsigned int *flags_list;

  comedi_range *rangeinfo;
  ccomedi_range **rangeinfo_list;

  unsigned int has_cmd;
  unsigned int has_insn_bits;

  int cmd_mask_errno;
  comedi_cmd *cmd_mask;
  int cmd_timed_errno;
  comedi_cmd *cmd_timed;
};
```

7.3.2. comedi_devinfo

The data type *comedi_devinfo* is used to store information about a device. This structure is usually filled in automatically when the driver is loaded (“attached”), so programmers need not access this data structure directly.

```
typedef struct comedi_devinfo_struct comedi_devinfo;

struct comedi_devinfo_struct{
  unsigned int version_code;    // version number of the Comedi code
  unsigned int n_subdevs;       // number of subdevices on this device
  char         driver_name[COMEDI_NAMELEN];
  char         board_name[COMEDI_NAMELEN];
  int          read_subdevice;  // number of read devices
  int          write_subdevice; // number of write devices
  int          unused[30];
};
```

7.3.3. comedi_t

The data type *comedi_t* is used to represent an open Comedi device:

```
typedef struct comedi_t_struct comedi_t;

struct comedi_t_struct{
  int magic;        // driver-specific magic number, for identification
  int fd;           // file descriptor, for open() and close()
  int n_subdevices; // number of subdevices on this device
  comedi_devinfo devinfo;
  subdevice *subdevices; // pointer to subdevice list
                         // filled in automatically at load time
  unsigned int has_insnlist_ioctl; // can process instruction lists
  unsigned int has_insn_ioctl;     // can process instructions
};
```

A valid *comedi_t* pointer is returned by a successful call to comedi_open(), and should be used for subsequent access to the device. It is a transparent type, and pointers to type *comedi_t* should not be dereferenced by the application.

7.3.4. sampl_t

```
typedef unsigned short sampl_t;
```

The data type sampl_t is one of the generic types used to represent data values in Comedilib. It is used in a few places where a data type shorter than lsampl_t is useful. On most architectures, sampl_t is defined to be *uint16*.

Most drivers represent data transferred by `read()` and `write()` using sampl_t. Applications should check the subdevice flag SDF_LSAMPL to determine if the subdevice uses sampl_t or lsampl_t.

7.3.5. lsampl_t

```
typedef unsigned int lsampl_t;
```

The data type lsampl_t is the data type typically used to represent data values in libcomedi. On most architectures, lsampl_t is defined to be uint32.

7.3.6. comedi_trig (deprecated)

```
typedef struct comedi_trig_struct comedi_trig;

struct comedi_trig_struct{
  unsigned int subdev;   /* subdevice */
  unsigned int mode;  /* mode */
  unsigned int flags;
  unsigned int n_chan;  /* number of channels */
  unsigned int *chanlist;   /* channel/range list */
  sampl_t *data;  /* data list, size depends on subd flags */
  unsigned int n;  /* number of scans */
  unsigned int trigsrc;
```

```
  unsigned int trigvar;
  unsigned int trigvar1;
  unsigned int data_len;
  unsigned int unused[3];
};
```

The comedi_trig structure is a control structure used by the COMEDI_TRIG ioctl, an older method of communicating instructions to the driver and hardware. Use of comedi_trig is deprecated, and should not be used in new applications.

7.3.7. comedi_sv_t

```
typedef struct comedi_sv_struct comedi_sv_t;

struct comedi_sv_struct{
  comedi_t *dev;
  unsigned int subdevice;
  unsigned int chan;

  /* range policy */
  int range;
  int aref;

  /* number of measurements to average (for ai) */
  int n;

  lsampl_t maxdata;
};
```

The comedi_sv_t structure is used by the comedi_sv_*() functions to provide a simple method of accurately measuring slowly varying inputs. See the relevant section for more details.

7.3.8. comedi_cmd

```
typedef struct comedi_cmd_struct comedi_cmd;

struct comedi_cmd_struct{
  unsigned int subdev;
  unsigned int flags;

  unsigned int start_src;
  unsigned int start_arg;

  unsigned int scan_begin_src;
  unsigned int scan_begin_arg;

  unsigned int convert_src;
  unsigned int convert_arg;

  unsigned int scan_end_src;
  unsigned int scan_end_arg;

  unsigned int stop_src;
```

```
  unsigned int stop_arg;

  unsigned int *chanlist;
  unsigned int chanlist_len;

  sampl_t *data;
  unsigned int data_len;
};
```

More information on using commands can be found in the command section.

7.3.9. comedi_insn

```
typedef struct comedi_insn_struct comedi_insn;

struct comedi_insn_struct{
  unsigned int insn;
  unsigned int n;
  lsampl_t*data;
  unsigned int subdev;
  unsigned int chanspec;
  unsigned int unused[3];
};
```

Comedi instructions are described by the comedi_insn structure. Applications send instructions to the driver in order to perform control and measurement operations that are done immediately or synchronously, i.e., the operations complete before program control returns to the application. In particular, instructions cannot describe acquisition that involves timers or external events.

The field insn determines the type of instruction that is sent to the driver. Valid instruction types are:

INSN_READ

: read values from an input channel

INSN_WRITE

: write values to an output channel

INSN_BITS

: read/write values on multiple digital I/O channels

INSN_CONFIG

: configure a subdevice

INSN_GTOD

: read a timestamp, identical to gettimeofday()

INSN_WAIT

: wait a specified number of nanoseconds

The number of samples to read or write, or the size of the configuration structure is specified by the field n, and the buffer for those samples by data. The field subdev is the subdevice index that the instruction is sent to. The field chanspec specifies the channel, range, and analog reference (if applicable).

Instructions can be sent to drivers using comedi_do_insn(). Multiple instructions can be sent to drivers in the same system call using comedi_do_insnlist().

7.3.10. comedi_range

```
typedef struct comedi_range_struct comedi_range;

struct comedi_range_struct{
  double min;
  double max;
  unsigned int unit;
}comedi_range;
```

The comedi_range structure conveys part of the information necessary to translate sample values to physical units, in particular, the endpoints of the range and the physical unit type. The physical unit type is specified by the field unit, which may take the values UNIT_volt for volts, UNIT_mA for milliamps, or UNIT_none for unitless. The endpoints are specified by the fields min and max.

7.3.11. comedi_krange

```
typedef struct comedi_krange_struct comedi_krange;

struct comedi_krange_struct{
  int min;
  int max;
  unsigned int flags;
};
```

The comedi_krange structure is used to transfer range information between the driver and Comedilib, and should not normally be used by applications. The structure conveys the same information as the comedi_range structure, except the fields min and max are integers, multiplied by a factor of 1000000 compared to the counterparts in comedi_range.

In addition, kcomedilib uses the comedi_krange structure in place of the comedi_range structure.

7.3.12. comedi_insnlist

```
typedef struct comedi_insnlist_struct comedi_insnlist;

struct comedi_insnlist_struct{
  unsigned int n_insns;
  comedi_insn *insns;
};
```

An instruction list (insnlist) structure is used to communicate a list of instructions.

7.4. Comedi Function Reference

comedi_close

Name

`comedi_close` — close a Comedi device

Synopsis

```
#include <comedilib.h>
int comedi_close(comedi * device);
```

Description

Close a device previously opened by comedi_open().

Return value

If sucessful, comedi_close returns 0. On failure, -1 is returned.

comedi_open

Name

`comedi_open` — open a Comedi device

Synopsis

```
#include <comedilib.h>
comedi_t * comedi_open(const char * filename);
```

Description

Open a Comedi device specified by the file filename.

Return value

If sucessful, comedi_open returns a pointer to a valid comedi_t structure. This structure is transparent; the pointer should not be dereferenced by the application. NULL is returned on failure.

comedi_loglevel

Name

`comedi_loglevel` — change Comedilib logging properties

Synopsis

```
#include <comedilib.h>
int comedi_loglevel(int loglevel);
```

Description

This function affects the output of debugging and error messages from Comedilib. By increasing the loglevel, additional debugging information will be printed. Error and debugging messages are printed to the stream stderr.

The default loglevel can be set by using the environment variable COMEDI_LOGLEVEL. The default loglevel is 1.

In order to conserve resources, some debugging information is disabled by default when Comedilib is compiled.

The meaning of the loglevels is as follows:

COMEDI_LOGLEVEL=0 Comedilib prints nothing.

COMEDI_LOGLEVEL=1 (default) Comedilib prints error messages when there is a self-consistency error (i.e., an internal bug.)

COMEDI_LOGLEVEL=2 Comedilib prints an error message when an invalid parameter is passed.

COMEDI_LOGLEVEL=3 Comedilib prints an error message whenever an error is generated in the Comedilib library or in the C library, when called by Comedilib.

COMEDI_LOGLEVEL=4 Comedilib prints a lot of junk.

Return value

This function returns the previous loglevel.

comedi_perror

Name

`comedi_perror` — print a Comedilib error message

Synopsis

```
#include <comedilib.h>
void comedi_perror(const char * s);
```

Description

When a Comedilib function fails, it usually returns -1 or NULL, depending on the return type. An internal library variable stores an error number, which can be retrieved with comedi_errno(). This error number can be converted to a human-readable form by the functions comedi_perror() and comedi_strerror().

These functions are intended to mimic the behavior of the standard C library functions perror(), strerror(), and errno. In particular, Comedilib functions sometimes return an error that is generated inside the C library; the comedi error message in this case is the same as the C library.

The function comedi_perror() prints an error message to stderr. The error message consists of the argument string, a colon, a space, a description of the error condition, and a new line.

comedi_strerror

Name

comedi_strerror — return string describing Comedilib error code

Synopsis

```
#include <comedilib.h>
char * comedi_strerror(int errnum);
```

Description

When a Comedilib function fails, it usually returns -1 or NULL, depending on the return type. An internal library variable stores an error number, which can be retrieved with comedi_errno(). This error number can be converted to a human-readable form by the functions comedi_perror() and comedi_strerror().

These functions are intended to mimic the behavior of the standard C library functions perror(), strerror(), and errno. In particular, Comedilib functions sometimes return an error that is generated inside the C library; the comedi error message in this case is the same as the C library.

The function comedi_strerror() returns a pointer to a character string describing the Comedilib error errnum. The persistence of the returned pointer is undefined, and should not be trusted after the next Comedilib call. An unrecognized error number will return a pointer to the string "undefined error", or similar.

comedi_errno

Name

comedi_errno — number of last Comedilib error

Synopsis

```
#include <comedilib.h>
int comedi_errno(void );
```

Description

When a Comedilib function fails, it usually returns -1 or NULL, depending on the return type. An internal library variable stores an error number, which can be retrieved with comedi_errno(). This error number can be converted to a human-readable form by the functions comedi_perror() and comedi_strerror().

These functions are intended to mimic the behavior of the standard C library functions perror(), strerror(), and errno. In particular, Comedilib functions sometimes return an error that is generated inside the C library; the comedi error message in this case is the same as the C library.

The function comedi_errno() returns an integer describing the most recent comedilib error. This integer may be used as the errnum parameter for comedi_strerror().

Note that comedi_errno() is deliberately different than the variable errno. This is to overcome difficulties in making errno thread-safe.

comedi_fileno

Name

`comedi_fileno` — integer descriptor of Comedilib device

Synopsis

```
#include <comedilib.h>
int comedi_fileno(comedi_t * device);
```

Description

The function comedi_fileno returns the integer descriptor for the device dev. This descriptor can then be used as the file descriptor parameter of read(), write(), etc. This function is intended to mimic the standard C library function fileno(). If dev is an invalid comedi_t pointer, the function returns -1 and sets the appropriate Comedilib error value.

comedi_get_n_subdevices

Name

`comedi_get_n_subdevices` — number of subdevices

Synopsis

```
#include <comedilib.h>
int comedi_get_n_subdevices(comedi_t * device);
```

Description

Returns the number of subdevices belonging to the Comedi device referenced by the parameter device.

comedi_get_version_code

Name

`comedi_get_version_code` — Comedi version code

Synopsis

```
#include <comedilib.h>
int comedi_get_version_code(comedi_t * device);
```

Description

Returns the Comedi kernel module version code. A valid Comedi device referenced by the parameter device is necessary to communicate with the kernel module. On error, -1 is returned.

The version code is encoded as a bitfield of three 8-bit numbers. For example, 0x00073d is the version code for version 0.7.61.

This function is of limited usefulness. A typical mis-application of this function is to use it to determine if a certain feature is supported. If the application needs to know of the existence of a particular feature, an existence test function should be written and put in the Comedilib source.

comedi_get_driver_name

Name

`comedi_get_driver_name` — Comedi driver name

Synopsis

```
#include <comedilib.h>
char * comedi_get_driver_name(comedi_t * device);
```

Description

The function comedi_get_driver_name returns a pointer to a string containing the name of the driver being used by comedi for the comedi device represented by device. This pointer is valid until the device is closed. This function returns NULL if there is an error.

comedi_get_board_name

Name

`comedi_get_board_name` — Comedi device name

Synopsis

```
#include <comedilib.h>
char * comedi_get_board_name(comedi_t * device);
```

Description

The function comedi_get_board_name returns a pointer to a string containing the name of the device. This pointer is valid until the comedi descriptor it is closed. This function returns NULL if there is an error.

comedi_get_subdevice_type

Name

`comedi_get_subdevice_type` — type of subdevice

Synopsis

```
#include <comedilib.h>
int comedi_get_subdevice_type(comedi_t * device, unsigned int subdevice);
```

Description

The function comedi_get_subdevice_type() returns an integer describing the type of subdevice that belongs to the comedi device device and has the index subdevice. The function returns -1 if there is an error. XXX Subdevice type table

comedi_find_subdevice_by_type

Name

`comedi_find_subdevice_by_type` — search for subdevice type

Synopsis

```
#include <comedilib.h>
int comedi_find_subdevice_by_type(comedi_t * device, int type, unsigned int
start_subdevice);
```

Description

The function comedi_find_subdevice_by_type() tries to locate a subdevice belonging to comedi device device, having type type, starting with the subdevice start_subdevice. If it finds a subdevice with the requested type, it returns its index. If it does not locate the requested subdevice, it returns -1 and sets the Comedilib error number to XXX "subdevice not found". If there is an error, the function returns -1 and sets the appropriate error.

comedi_get_read_subdevice

Name

`comedi_get_read_subdevice` — find streaming input subdevice

Synopsis

```
#include <comedilib.h>
int comedi_get_read_subdevice(comedi_t * device);
```

Description

The function comedi_get_read_subdevice() returns the subdevice that allows streaming input for device dev. If no subdevice supports streaming input, -1 is returned and the Comedilib error number is set to XXX "subdevice not found".

comedi_get_write_subdevice

Name

`comedi_get_write_subdevice` — find streaming output subdevice

Synopsis

```
#include <comedilib.h>
int comedi_get_write_subdevice(comedi_t * device);
```

Description

The function comedi_get_write_subdevice() returns the subdevice that allows streaming output for device dev. If no subdevice supports streaming output, -1 is returned and the Comedilib error number is set to XXX "subdevice not found".

comedi_get_subdevice_flags

Name

`comedi_get_subdevice_flags` — properties of subdevice

Synopsis

```
#include <comedilib.h>
int comedi_get_subdevice_flags(comedi_t * device, unsigned int subdevice);
```

Description

This function returns a bitfield describing the capabilities of the specified subdevice. If there is an error, -1 is returned, and the Comedilib error value is set.

XXX table.

comedi_get_n_channels

Name

`comedi_get_n_channels` — number of subdevice channels

Synopsis

```
#include <comedilib.h>
int comedi_get_n_channels(comedi_t * device, unsigned int subdevice);
```

Description

The function comedi_get_n_channels() returns the number of channels of the subdevice belonging to the comedi device device and having index subdevice. This function returns -1 on error and the Comedilib error value is set.

comedi_range_is_chan_specific

Name

comedi_range_is_chan_specific — range information depends on channel

Synopsis

```
#include <comedilib.h>
int comedi_range_is_chan_specific(comedi_t * device, unsigned int subdevice);
```

Description

If each channel of the specified subdevice has different range information, this function returns 1. Otherwise, this function returns 0. On error, this function returns -1.

comedi_maxdata_is_chan_specific

Name

comedi_maxdata_is_chan_specific — maximum sample depends on channel

Synopsis

```
#include <comedilib.h>
int comedi_maxdata_is_chan_specific(comedi_t * device, unsigned int subdevice);
```

Description

If each channel of the specified subdevice has different maximum sample values, this function returns 1. Otherwise, this function returns 0. On error, this function returns -1.

comedi_get_maxdata

Name

`comedi_get_maxdata` — maximum sample of channel

Synopsis

```
#include <comedilib.h>
lsampl_t comedi_get_maxdata(comedi_t * device, unsigned int subdevice, unsigned int
channel);
```

Description

The function comedi_get_maxdata() returns the maximum valid data value for channel chan of subdevice subdevice belonging to the comedi device device This function returns 0 on error.

comedi_get_n_ranges

Name

`comedi_get_n_ranges` — number of ranges of channel

Synopsis

```
#include <comedilib.h>
int comedi_get_n_ranges(comedi_t * device, unsigned int subdevice, unsigned int
channel);
```

Description

The function comedi_get_n_ranges() returns the number of ranges of the channel chan belonging to the subdevice of the comedi device device. This function returns -1 on error.

comedi_get_range

Name

comedi_get_range — range information of channel

Synopsis

```
#include <comedilib.h>
comedi_range * comedi_get_range(comedi_t * device, unsigned int subdevice, unsigned
int channel, unsigned int range);
```

Description

The function comedi_get_range() returns a pointer to a comedi_range structure that contains information that can be used to convert sample values to or from physical units. The pointer is valid until the Comedi device device is closed. If there is an error, NULL is returned.

comedi_find_range

Name

comedi_find_range — search for range

Synopsis

```
#include <comedilib.h>
int comedi_find_range(comedi_t * device, unsigned int subdevice, unsigned int
channel, unsigned int unit, double min, double max);
```

Description

The function comedi_find_range() tries to locate the optimal (smallest) range for the channel chan belonging to a subdevice of the comedi device device, that includes both min and max in units. If a matching range is found, the index of the matching range is returned. If no matching range is available, the function returns -1.

comedi_get_buffer_size

Name

comedi_get_buffer_size — streaming buffer size of subdevice

Synopsis

```
#include <comedilib.h>
int comedi_get_buffer_size(comedi_t * device, unsigned int subdevice);
```

Description

The function comedi_get_buffer_size() returns the size (in bytes) of the streaming buffer for the subdevice specified by device and subdevice. On error, -1 is returned.

comedi_get_max_buffer_size

Name

comedi_get_max_buffer_size — maximum streaming buffer size

Synopsis

```
#include <comedilib.h>
int comedi_get_max_buffer_size(comedi_t * device, unsigned int subdevice);
```

Description

The function comedi_get_max_buffer_size() returns the maximum allowable size (in bytes) of the streaming buffer for the subdevice specified by device and subdevice. Changing the maximum buffer size requires appropriate privileges. On error, -1 is returned.

comedi_set_buffer_size

Name

comedi_set_buffer_size — streaming buffer size of subdevice

Synopsis

```
#include <comedilib.h>
int comedi_set_buffer_size(comedi_t * device, unsigned int subdevice, unsigned int
size);
```

Description

The function comedi_set_buffer_size() changes the size of the streaming buffer for the subdevice specified by device and subdevice. The parameter size must be a multiple of the virtual memory page size.

The virtual memory page size can be determined using sysconf(_SC_PAGE_SIZE).

comedi_trigger

Name

`comedi_trigger` — perform streaming input/output (deprecated)

Synopsis

```
#include <comedilib.h>
int comedi_trigger(comedi_t * device, comedi_trig * trig);
```

Status

deprecated

Description

The function comedi_trigger() instructs Comedi to perform the command specified by the trigger structure trig. The return value depends on the particular trig being issued. If there is an error, -1 is returned.

comedi_do_insnlist

Name

`comedi_do_insnlist` — perform multiple instructions

Synopsis

```
#include <comedilib.h>
int comedi_do_insnlist(comedi_t * device, comedi_insnlist * list);
```

Description

The function comedi_do_insnlist() performs multiple Comedi instructions as part of one system call. In addition, Comedi attempts to perform the instructions atomically, that is, on standard Linux kernels, no process preemption should occur during the instructions. However, the process may be preempted before or after the group of instructions.

This function can be used to avoid the overhead of multiple system calls, or to ensure that multiple instructions occur without significant delay between them. Preemption may occur if any of the instructions or the data arrays of any of the instructions exist in non-resident or copy-on-write pages.

Return value

The function comedi_do_insnlist() returns the number of sucessfully completed instructions. Error information for the unsucessful instruction is not available. If there is an error before the first instruction can be executed, -1 is returned.

comedi_do_insn

Name

`comedi_do_insn` — perform instruction

Synopsis

```
#include <comedilib.h>
int comedi_do_insn(comedi_t * device, comedi_insn * instruction);
```

Description

The function comedi_do_insn() performs a single instruction. If sucessful, comedi_do_insn() returns the number of samples measured, which may be less than the number of requested samples. Comedi limits the number of requested samples in order to enforce fairness among processes. If there is an error, -1 is returned.

comedi_lock

Name

comedi_lock — subdevice reservation

Synopsis

```
#include <comedilib.h>
int comedi_lock(comedi_t * device, unsigned int subdevice);
```

Description

The function comedi_lock() reserves a subdevice for use by the current process. While the lock is held, no other process is allowed to read, write, or configure that subdevice, although other processes can read information about the subdevice. The lock is released when comedi_unlock() is called, or the device is closed. If sucessful, 0 is returned. If there is an error, -1 is returned.

comedi_unlock

Name

comedi_unlock — subdevice reservation

Synopsis

```
#include <comedilib.h>
int comedi_unlock(comedi_t * device, unsigned int subdevice);
```

Description

The function comedi_unlock() released a subdevice lock acquired by comedi_lock(). If sucessful, 0 is returned, otherwise -1.

comedi_to_phys

Name

`comedi_to_phys` — convert sample to physical units

Synopsis

```
#include <comedilib.h>
double comedi_to_phys(lsampl_t data, comedi_range * range, lsampl_t maxdata);
```

Description

Converts data given in sample values (lsampl_t, between 0 and maxdata) into physical units (double). The parameter range represents the conversion information to use, and the parameter maxdata represents the maximum possible data value for the channel that the data was read.

Conversion of endpoint sample values, that is, sample values equal to 0 or maxdata, is affected by the Comedilib out-of-range behavior. If the out-of-range behavior is set to COMEDI_OOR_NAN, endpoint values are converted to NAN. If the out-of-range behavior is set to COMEDI_OOR_NUMBER, the endpoint values are converted similarly to other values.

If there is an error, NAN is returned.

comedi_from_phys

Name

`comedi_from_phys` — convert physical units to sample

Synopsis

```
#include <comedilib.h>
lsampl_t comedi_from_phys(double data, comedi_range * range, lsampl_t maxdata);
```

Description

Converts data given in physical units (data) into sample values (lsampl_t, between 0 and maxdata). The parameter rng represents the conversion information to use, and the parameter maxdata represents the maximum possible data value for the channel that the data will be written to.

Conversion is not affected by out-of-range behavior. Out-of-range data parameters are silently truncated to the range 0 to maxdata.

comedi_data_read

Name

`comedi_data_read` — read single sample from channel

Synopsis

```
#include <comedilib.h>
int comedi_data_read(comedi_t * device, unsigned int subdevice, unsigned int
channel, unsigned int range, unsigned int aref, lsampl_t * data);
```

Description

Reads a single sample on the channel specified by the Comedi device device, the subdevice subdevice, and the channel channel. For the A/D conversion (if appropriate), the device is configured to use range specification range and (if appropriate) analog reference type aref. Analog reference types that are not supported by the device are silently ignored.

The function comedi_data_read() reads one data value from the specified channel and places the data value in the location pointed to by data.

WARNING: comedi_data_read() does not do any pausing to allow multiplexed analog inputs to settle before performing an analog to digital conversion. If you are switching between different channels and need to allow your analog input to settle for an accurate reading, use comedi_data_read_delayed(), or set the input channel at an earlier time with comedi_data_read_hint().

On sucess, comedi_data_read() returns 1 (the number of samples read). If there is an error, -1 is returned.

Data values returned by this function are unsigned integers less than or equal to the maximum sample value of the channel, which can be determined using the function comedi_get_maxdata(). Conversion of data values to physical units can be performed by the function comedi_to_phys().

comedi_data_read_delayed

Name

`comedi_data_read_delayed` — read single sample from channel after delaying for specified settling time

Synopsis

```
#include <comedilib.h>
int comedi_data_read_delayed(comedi_t * device, unsigned int subdevice, unsigned
int channel, unsigned int range, unsigned int aref, lsampl_t * data, unsigned int
nanosec);
```

Description

Similar to comedi_data_read() except it will wait for the specified number of nanoseconds between setting the input channel and taking a sample. For analog inputs, most boards have a single analog to digital converter which is multiplexed to be able to read multiple channels. If the input is not allowed to settle after the multiplexer switches channels, the reading will be inaccurate. This function is useful for allowing a multiplexed analog input to settle when switching channels.

Although the settling time is specified in nanoseconds, the actual settling time will be rounded up to the nearest microsecond.

comedi_data_read_hint

Name

`comedi_data_read_hint` — tell driver which channel/range/aref you are going to read from next

Synopsis

```
#include <comedilib.h>
int comedi_data_read_hint(comedi_t * device, unsigned int subdevice, unsigned int
channel, unsigned int range, unsigned int aref);
```

Description

Used to prepare an analog input for a subsequent call to comedi_data_read(). It is not necessary to use this function, but it can be useful for eliminating inaccuaracies caused by insufficient settling times when switching the channel or gain on an analog input. This function sets an analog input to the channel, range, and aref specified but does not perform an actual analog to digital conversion.

Alternatively, one can simply use comedi_data_read_delayed(), which sets up the input, pauses to allow settling, then performs a conversion.

comedi_data_write

Name

`comedi_data_write` — write single sample to channel

Synopsis

```
#include <comedilib.h>
int comedi_data_write(comedi_t * device, unsigned int subdevice, unsigned int
channel, unsigned int range, unsigned int aref, lsampl_t data);
```

Description

Writes a single sample on the channel that is specified by the Comedi device device, the subdevice subdevice, and the channel channel. If appropriate, the device is configured to use range specification range and analog reference type aref. Analog reference types that are not supported by the device are silently ignored.

The function comedi_data_write() writes the data value specified by the parameter data to the specified channel.

On sucess, comedi_data_write() returns 1 (the number of samples written). If there is an error, -1 is returned.

comedi_dio_config

Name

`comedi_dio_config` — change input/output properties of channel

Synopsis

```
#include <comedilib.h>
int comedi_dio_config(comedi_t * device, unsigned int subdevice, unsigned int
channel, unsigned int direction);
```

Description

The function comedi_dio_config() configures individual channels in a digital I/O subdevice to be either input or output, depending on the value of parameter direction. Valid directions are COMEDI_INPUT or COMEDI_OUTPUT.

Depending on the capabilities of the hardware device, multiple channels may be grouped together to determine direction. In this case, a single call to comedi_dio_config() for any channel in the group will affect the entire group.

If sucessful, 1 is returned, otherwise -1.

comedi_dio_read

Name

`comedi_dio_read` — read single bit from digital channel

Synopsis

```
#include <comedilib.h>
int comedi_dio_read(comedi_t * device, unsigned int subdevice, unsigned int
channel, unsigned int * bit);
```

Description

The function reads the channel channel belonging to the subdevice subdevice of device device. The data value that is read is stored in the location pointed to by bit. This function is equivalent to comedi_data_read(device,subdevice,channel,0,0,bit). This function does not require a digital subdevice or a subdevice with a maximum data value of 1 to work properly.

Return values and errors are the same as comedi_data_read().

comedi_dio_write

Name

`comedi_dio_write` — write single bit to digital channel

Synopsis

```
#include <comedilib.h>
int comedi_dio_write(comedi_t * device, unsigned int subdevice, unsigned int
channel, unsigned int bit);
```

Description

The function writes the value bit to the channel channel belonging to the subdevice subdevice of device device. This function is equivalent to comedi_data_write(device,subdevice,channel,0,0,bit). This function does not require a digital subdevice or a subdevice with a maximum data value of 1 to work properly.

Return values and errors are the same as comedi_data_write().

comedi_dio_bitfield

Name

comedi_dio_bitfield — read/write multiple digital channels

Synopsis

```
#include <comedilib.h>
int comedi_dio_bitfield(comedi_t * device, unsigned int subdevice, unsigned int
write_mask, unsigned int * bits);
```

Description

The function comedi_dio_bitfield() allows multiple channels to be read simultaneously from a digital input or digital I/O device. The parameter write_mask and the value pointed to by bits are interpreted as bit fields, with the least significant bit representing channel 0. For each bit in write_mask that is set to 1, the cooresponding bit in *bits is written to the digital output channel. After writing all the output channels, each channel is read, and the result placed in the approprate bits in *bits. The result of reading an output channel is undefined. It is not possible to access channels greater than 31 using this function.

comedi_sv_init

Name

comedi_sv_init — slowly-varying inputs

Synopsis

```
#include <comedilib.h>
int comedi_sv_init(comedi_sv_t * sv, comedi_t * device, unsigned int subdevice,
unsigned int channel);
```

Status

deprecated

Description

The function comedi_sv_init() initializes the slow varying Comedi structure sv to use the device device, the analog input subdevice subdevice, and the channel channel. The slow varying Comedi structure is used by comedi_sv_measure() to accurately measure an analog input by averaging over many samples. The default number of samples is 100. This function returns 0 on success, -1 on error.

comedi_sv_update

Name

`comedi_sv_update` — slowly-varying inputs

Synopsis

```
#include <comedilib.h>
int comedi_sv_update(comedi_sv_t * sv);
```

Status

deprecated

Description

The function comedi_sv_update() updates internal parameters of the slowly varying Comedi structure sv.

comedi_sv_measure

Name

`comedi_sv_measure` — slowly-varying inputs

Synopsis

```
#include <comedilib.h>
int comedi_sv_measure(comedi_sv_t * sv, double * data);
```

Status

deprecated

Description

The function comedi_sv_measure() uses the slowly varying Comedi structure sv to measure a slowly varying signal. If sucessful, the result (in physical units) is stored in the location pointed to by data, and the number of samples is returned. On error, -1 is returned.

comedi_get_cmd_src_mask

Name

`comedi_get_cmd_src_mask` — streaming input/output capabilities

Synopsis

```
#include <comedilib.h>
int comedi_get_cmd_src_mask(comedi_t * device, unsigned int subdevice, comedi_cmd *
command);
```

Description

The command capabilities of the subdevice indicated by the parameters device and subdevice are probed, and the results placed in the command structure pointed to by the parameter command. The trigger source elements of the command structure are set to the logical OR value of possible trigger sources. Other elements in the structure are undefined. If sucessful, 0 is returned, otherwise -1.

comedi_get_cmd_generic_timed

Name

comedi_get_cmd_generic_timed — streaming input/output capabilities

Synopsis

```
#include <comedilib.h>
int comedi_get_cmd_generic_timed(comedi_t * device, unsigned int subdevice,
comedi_cmd * command, unsigned int period_ns);
```

Description

The command capabilities of the subdevice indicated by the parameters device and subdevice are probed, and the results placed in the command structure pointed to by the parameter command. The command structure pointed to by the parameter command is modified to be a valid command that can be used as a parameter to comedi_command(). The command measures samples at a rate that corresponds to the period period_ns. The rate is adjusted to a rate that the device can handle. If sucessful, 0 is returned, otherwise -1.

comedi_cancel

Name

comedi_cancel — stop streaming input/output in progress

Synopsis

```
#include <comedilib.h>
int comedi_cancel(comedi_t * device, unsigned int subdevice);
```

Description

The function comedi_cancel() can be used to stop a Comedi command previously started by comedi_command() that is still in progress on the subdevice indicated by the parameters device and subdevice. This may not return the subdevice to a ready state, since there may be samples in the buffer that need to be read.

If sucessful, 0 is returned, otherwise -1.

comedi_command

Name

`comedi_command` — start streaming input/output

Synopsis

```
#include <comedilib.h>
int comedi_command(comedi_t * device, comedi_cmd * command);
```

Description

The function comedi_command() starts streaming input or output. The command structure pointed to by the parameter command specifies the acquisition. The command must be able to pass comedi_command_test() with a return value of 0, or comedi_command() will fail. For input subdevices, sample values are read using the function read(). For output subdevices, sample values are written using the function write().

If sucessful, 0 is returned, otherwise -1.

comedi_command_test

Name

`comedi_command_test` — test streaming input/output configuration

Synopsis

```
#include <comedilib.h>
int comedi_command_test(comedi_t * device, comedi_cmd * command);
```

Description

The function comedi_command_test() tests the command structure pointed to by the parameter command and returns an integer describing the testing stages that were sucessfully passed. In addition, if elements of the command structure are invalid, they may be modified. Source elements are modified to remove invalid source triggers. Argument elements are adjusted or rounded to the nearest valid value.

The meanings of the return value are as follows.

0 indicates a valid command.

1 indicates that one of the *_src members of the command contained an unsupported trigger. The bits corresponding to the unsupported triggers are zeroed.

2 indicates that the particular combination of *_src settings is not supported by the driver, or that one of the *_src members has the bit corresponding to multiple trigger sources set at the same time.

3 indicates that one of the *_arg members of the command is set outside the range of allowable values. For instance, an argument for a TRIG_TIMER source which exceeds the board's maximum speed. The invalid *_arg members will be adjusted to valid values.

4 indicates that one of the *_arg members required adjustment. For instance, the argument of a TRIG_TIMER source may have been rounded to the nearest timing period supported by the board.

5 indicates that some aspect of the command's chanlist is unsupported by the board. For example, some board's require that all channels in the chanlist use the same range.

comedi_poll

Name

`comedi_poll` — force updating of streaming buffer

Synopsis

```
#include <comedilib.h>
int comedi_poll(comedi_t * device, unsigned int subdevice);
```

Description

The function comedi_poll() is used on a subdevice that has a Comedi command in progress in order to update the streaming buffer. If supported by the driver, all available samples are copied to the streaming buffer. These samples may be pending in DMA buffers or device FIFOs. If sucessful, the number of additional bytes available is returned. If there is an error, -1 is returned.

comedi_set_max_buffer_size

Name

`comedi_set_max_buffer_size` — streaming buffer size of subdevice

Synopsis

```
#include <comedilib.h>
int comedi_set_max_buffer_size(comedi_t * device, unsigned int subdevice, unsigned
int max_size);
```

Description

The function comedi_set_max_buffer_size() changes the maximum allowable size (in bytes) of the streaming buffer for the subdevice specified by device and subdevice. Changing the maximum buffer size requires appropriate privileges. If sucessful, the old buffer size is returned. On error, -1 is returned.

comedi_get_buffer_contents

Name

`comedi_get_buffer_contents` — streaming buffer status

Synopsis

```
#include <comedilib.h>
int comedi_get_buffer_contents(comedi_t * device, unsigned int subdevice);
```

Description

The function comedi_get_buffer_contents() is used on a subdevice that has a Comedi command in progress. The number of bytes that are available in the streaming buffer is returned. If there is an error, -1 is returned.

comedi_mark_buffer_read

Name

`comedi_mark_buffer_read` — streaming buffer status

Synopsis

```
#include <comedilib.h>
int comedi_mark_buffer_read(comedi_t * device, unsigned int subdevice, unsigned int
num_bytes);
```

Description

The function comedi_mark_buffer_read() is used on a subdevice that has a Comedi command in progress. This function can be used to indicate that the next num_bytes bytes in the buffer are no longer needed and may be discarded. If there is an error, -1 is returned.

comedi_get_buffer_offset

Name

`comedi_get_buffer_offset` — streaming buffer status

Synopsis

```
#include <comedilib.h>
int comedi_get_buffer_offset(comedi_t * device, unsigned int subdevice);
```

Description

The function comedi_get_buffer_offset() is used on a subdevice that has a Comedi command in progress. This function returns the offset in bytes of the read pointer in the streaming buffer. This offset is only useful for memory mapped buffers. If there is an error, -1 is returned.

comedi_get_timer

Name

`comedi_get_timer` — timer information (deprecated)

Synopsis

```
#include <comedilib.h>
int comedi_get_timer(comedi_t * device, unsigned int subdevice, double frequency,
unsigned int * trigvar, double * actual_frequency);
```

Status

deprecated

Description

The function comedi_get_timer converts the frequency frequency to a number suitable to send to the driver in a comedi_trig structure. This function remains for compatibility with very old versions of Comedi, that converted sampling rates to timer values in the libary. This conversion is now done in the kernel, and every device has the timer type nanosec_timer, indicating that timer values are simply a time specified in nanoseconds.

comedi_timed_1chan

Name

`comedi_timed_1chan` — streaming input (deprecated)

Synopsis

```
#include <comedilib.h>
int comedi_timed_1chan(comedi_t * device, unsigned int subdevice, unsigned int
channel, unsigned int range, unsigned int aref, double frequency, unsigned int
num_samples, double * data);
```

Status

deprecated

Description

Not documented.

comedi_set_global_oor_behavior

Name

`comedi_set_global_oor_behavior` — out-of-range behavior

Synopsis

```
#include <comedilib.h>
int comedi_set_global_oor_behavior(enum comedi_oor_behavior behavior);
```

Status

alpha

Description

This function changes the Comedilib out-of-range behavior. This currently affects the behavior of comedi_to_phys() when converting endpoint sample values, that is, sample values equal to 0 or maxdata. If the out-of-range behavior is set to COMEDI_OOR_NAN, endpoint values are converted to NAN. If the out-of-range behavior is set to COMEDI_OOR_NUMBER, the endpoint values are converted similarly to other values.

The previous out-of-range behavior is returned.

comedi_apply_calibration

Name

`comedi_apply_calibration` — set calibration from file

Synopsis

```
#include <comedilib.h>
int comedi_apply_calibration(comedi_t *device, unsigned int subdevice, unsigned int
channel, unsigned int range, unsigned int aref, const char *file_path);
```

Status

alpha

Description

This function sets the calibration of the specified subdevice so that it is in proper calibration when using the specified channel, range and aref. It does so by performing writes to the appropriate channels of the board's calibration subdevice(s). Depending on the hardware, the calibration settings used may or may not depend on the channel, range, or aref. Furthermore, the calibrations for different channels, ranges, or arefs may not be independent. For example, some boards cannot have their analog inputs calibrated for multiple input ranges simultaneously. Applying a calibration for range 1 may blow away a previously applied calibration for range 0. Applying a calibration for analog input channel 0 may cause the same calibration to be applied to all the other analog input channels as well. Your only guarantee is that calls to comedi_apply_calibration() on different subdevices will not interfere with each other.

In practice, their are some rules of thumb on how calibrations behave. No calibrations depend on the aref. A multiplexed analog input will have calibration settings that do not depend on the channel, and applying a setting for one channel will affect all channels equally. Analog outputs, and analog inputs with independent a/d converters for each input channel, will have calibrations settings which do depend on the channel, and the settings for each channel will be independent of the other channels.

If you wish to investigate exactly what comedi_apply_calibration() is doing, you can perform reads on your board's calibration subdevice to see which calibration channels it is changing. You can also try to decipher the calibration file directly (it's a text file).

The file_path parameter can be used to specify the file which contains the calibration information. If *`file_path`* is NULL, then comedilib will use a default file location. The calibration information used by this function is generated by the comedi_calibrate program (see its man page).

The functions comedi_parse_calibration_file(), comedi_apply_parsed_calibration(), and comedi_cleanup_calibration() provide the same functionality at a slightly lower level.

Return value

Zero on success, a negative number on failure.

comedi_apply_parsed_calibration

Name

`comedi_apply_parsed_calibration` — set calibration from memory

Synopsis

```
#include <comedilib.h>
int comedi_apply_parsed_calibration(comedi_t * device, unsigned int subdevice,
unsigned int channel, unsigned int range, unsigned int aref, const
comedi_calibration_t *calibration);
```

Status

alpha

Description

This function is similar to comedi_apply_calibration() except the calibration information is read from memory instead of a file. This function can be more efficient than comedi_apply_calibration() since the calibration file does not need to be reparsed with every call. The *calibration* is obtained by a call to comedi_parse_calibration_file().

Return value

Zero on success, a negative number on failure.

comedi_cleanup_calibration_file

Name

comedi_cleanup_calibration_file — free calibration resources

Synopsis

```
#include <comedilib.h>
void comedi_cleanup_calibration_file(comedi_calibration_t *calibration);
```

Status

alpha

Description

This function frees the resources associated with a *calibration* obtained from comedi_parse_calibration_file(). *calibration* can not be used again after calling this function.

comedi_get_default_calibration_path

Name

comedi_get_default_calibration_path — get default calibration file path

Synopsis

```
#include <comedilib.h>
char* comedi_get_default_calibration_path(comedi_t *dev);
```

Status

alpha

Description

This function returns a string containing a default calibration file path appropriate for *dev*. Memory for the string is allocated by the function, and should be freed when the string is no longer needed.

Return value

A string which contains a file path useable by comedi_parse_calibration_file(). On error, NULL is returned.

comedi_parse_calibration_file

Name

comedi_parse_calibration_file — set calibration

Synopsis

```
#include <comedilib.h>
comedi_calibration_t* comedi_parse_calibration_file(const char *file_path);
```

Status

alpha

Description

This function parses a calibration file (produced by the comedi_calibrate program) and returns a pointer to a comedi_calibration_t which can be passed to the comedi_apply_parsed_calibration() function. When you are finished using the comedi_calibration_t, you should call comedi_cleanup_calibration() to free the resources associated with the comedi_calibration_t.

The comedi_get_default_calibration_path() function may be useful in conjunction with this function.

Return value

A pointer to parsed calibration information on success, or NULL on failure.

Glossary

Application Program Interface

> The (documented) set of function calls supported by a particular application, by which programmers can access the functionality available in the application.

buffer

> Comedi uses permanently allocated kernel memory for streaming input and output to store data that has been measured by a device, but has not been read by an application. These buffers can be resized by the Comedilib function `comedi_buffer_XXX()` or the `comedi_config` utility.

buffer overflow

> This is an error message that indicates that the driver ran out of space in a Comedi buffer to put samples. It means that the application is not copying data out of the buffer quickly enough. Often, this problem can be fixed by making the Comedi buffer larger. See `comedi_buffer_XXX` for more information.

Differential IO

...

Direct Memory Access

DMA is a method of transferring data between a device and the main memory of a computer. DMA operates differently on ISA and PCI cards. ISA DMA is handled by a controller on the motherboard and is limited to transfers to/from the lowest 16 MB of physical RAM and can only handle a single segment of memory at a time. These limitations make it almost useless. PCI ("bus mastering") DMA is handled by a controller on the device, and can typically address 4 GB of RAM and handle many segments of memory simultaneously. DMA is usually not the only means to data transfer, and may or may not be the optimal transfer mechanism for a particular situation.

First In, First Out

Most devices have a limited amount of on-board space to store samples before they are transferred to the Comedi buffer. This allows the CPU or DMA controller to do other things, and then efficiently process a large number of samples simultaneously. It also increases the maximum interrupt latency that the system can handle without interruptions in data.

Comedi command

Comedi commands are the mechanism that applications configure subdevices for streaming input and output.

command

See: Comedi command

configuration option

instruction

Comedi instructions are the mechanism used by applications to do immediate input from channels, output to channels, and configuration of subdevices and channels.

instruction list

Instruction lists allow the application to perform multiple Comedi instructions in the same system call.

option

See Also: option list .

option list

Option lists are used with `comedi_config` to perform driver configuration.
See Also: configuration option , option .

overrun

This is an error message that indicates that there was device-level problem, typically with trigger pulses occurring faster than the board can handle.

poll

The term poll (and polling) is used for several different related concepts in Comedi. Comedi implements the `poll()` system call for Comedi devices, which is similar to `select()`, and returns information about file descriptors that can be read or written. Comedilib also has a function called `comedi_poll()`, which causes the driver to copy all available data from the device to the Comedi buffer. In addition, some drivers may use a polling technique in place of interrupts.