Comedi

The Control and Measurement Device Interface handbook for Comedilib 0.11.0

Contents

List of Figures

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 source code is distributed in two main packages, comedi and comedilib:

- **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 use by *real-time* kernel modules. 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 and RTLinux/GPL.

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 issues in mind:

- **Kernel space vs. User space.** The Linux operating system has two levels that require 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 the *Comedilib* user-space library. *Comedilib* then handles the necessary communication with the *Comedi* modules running in kernel-space.
- **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. Each device supported in the kernel may be representated as such a user space device file, and its functionality can may be accessed by classical Unix file I/O: `open()`, `close()`, `read()`, `write()`, `ioctl()`, and `mmap()`.
- **`/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/GPL or RTAI application, there are a few extra requirements, because not all system calls are available in the kernel of the real-time operating systems RTLinux/GPL or RTAI. 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, counters 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 may also allow automatically sampling of a set of channels in a prescribed order.
- **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 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.
- **Subdevice**: a set of functionally identical channels. For example, a set of 16 identical analog inputs.
- **Device**: a set of subdevices 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 subdevice with 16 analog input channels, another subdevice with two analog output channels, and a third subdevice with eight digital inputs/outputs.

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 Comedi "commands", which are streaming asyncronous acquisitions. Figure 1 depicts a typical acquisition sequence when running a command:

- 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, some hardware has limits on the minimum **conversion interval** it can achieve, i.e., the minimum time it needs between *subsequent* conversions. For example, some A/D hardware must *multiplex* the conversions from different input channels onto one single A/D converter. Thus the conversions are done serially in time (as shown in Figure 1). Other cards have the hardware to do two or more acquisitions in parallel, and can perform all the conversions in a scan simultaneously. The begin of each conversion is "triggered" by some internally or externally generated pulse, e.g., a timer.

In general, not only the start of a *conversion* is triggered, but also the start 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 you to specify the **channel list**, i.e., the sequence of channels that needs to be acquired during each scan.

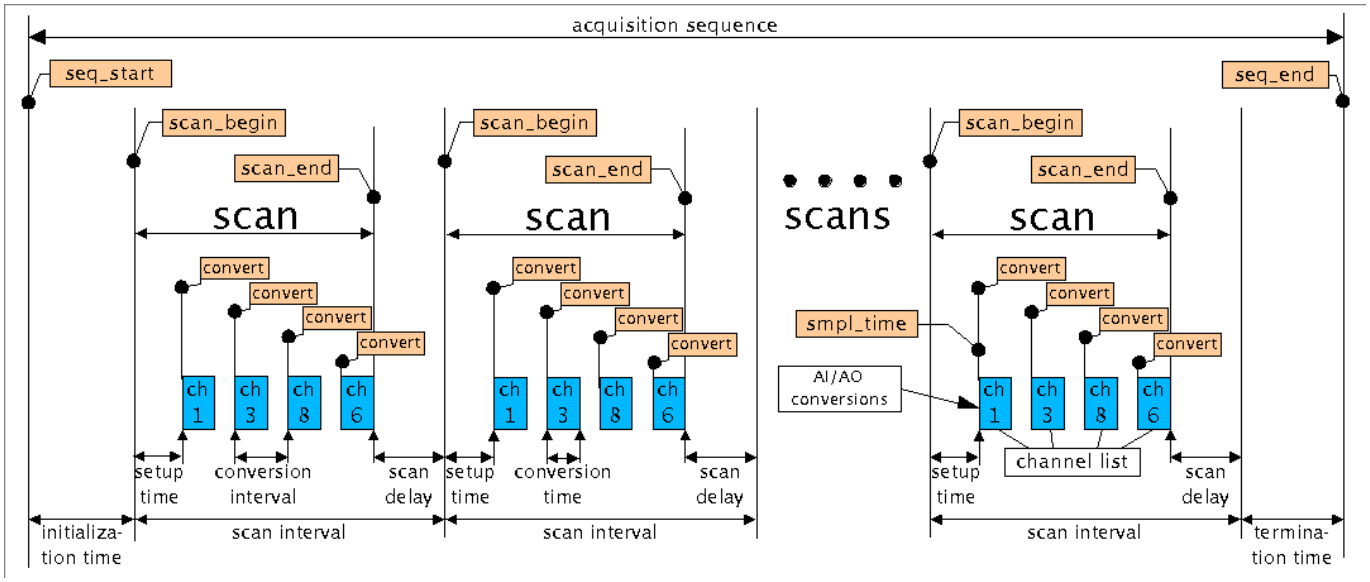

Figure courtesy of Kurt Müller.

Figure 1: Asynchronous Acquisition Sequence

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_read_delayed`(), `comedi_data_write`(), `comedi_dio_read`(), `comedi_dio_write`(). In addition, the lower-level `comedi_do_insn`() function can be used to perform an acquisition.

  "Synchronous" means that the calling process blocks until the data acquisition has finished.

- **Mutiple synchronous acquisition**: The `comedi_data_read_n`() function performs (possibly multiple) data acquisitions on a specified channel, in a **synchronous** way. So, the function call blocks until the whole acquisition has finished. The precise timing between the acquisitions is not hardware controlled.

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

- **Command**: a command is *sequence* of *scans*, for which conditions have been specified that determine when the acquisition will start and stop, and when each conversion in each scan should occur. A `comedi_command`() function call sets up the **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 timing of the data acquisition as it proceeds.

1.8 Supporting functionality

The command functionality cannot be offered by DAQ cards that lack the hardware to autonomously sequence a series of scans. For these cards, the command functionality may 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. 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 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. Comedi provides internal buffers for data being streamed to/from devices via Comedi commands. The buffer sizes are user-adjustable.

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, any jumper settings related to input ranges, the I/O base address and IRQ for old non-plug-n-play boards, etc.

2.1 Configuration

The good news is: on most systems PCI and USB based boards are configured automatically. The kernel will detect your data acquisition devices, will load the appropriate kernel drivers and will create the `/dev/comedi` entries.

```
bp1@bp1-x61:~/sandbox/comedilib$ ls -l /dev/comedi0*
crw-rw---- 1 root iocard 98,  0 2012-04-26 23:41 /dev/comedi0
crw-rw---- 1 root iocard 98, 48 2012-04-26 23:41 /dev/comedi0_subd0
crw-rw---- 1 root iocard 98, 49 2012-04-26 23:41 /dev/comedi0_subd1
```

Usually these devices belong to the group `iocard` as shown here. The only action you need to take is to become member of this group and then the Comedi device is ready to be used.

There are a few PCI drivers that for one reason or another do not support auto-configuration, either because there is more than one variant of a board sharing the same PCI device ID (e.g. Advantech PCI-1710 and PCI-1710HG), or because some configuration options are needed (e.g. Amplicon PCI224 and PCI234) or for some other reason. It is also possible to disable auto-configuration when loading the `comedi` kernel module. In these cases devices need to be configured manually as for ISA cards. Conversely, most Comedi drivers supplied with the kernel sources that support auto-configuration may no longer support manual configuration.

By default, the `comedi` kernel module does not reserve any devices for manual configuration so manual configuration will fail. To allow devices to be configured manually, set the *`comedi_num_legacy_minors`* module parameter to the number of devices to reserve for manual configuration when loading the `comedi` kernel module. If using **modprobe**, this can be set automatically by editing `/etc/modprobe.conf` or `/etc/modprobe.d/comedi.conf` (depending on the system) to include the line:

```
        options comedi comedi_num_legacy_minors=4
```

The number `4` in the above line may be adjusted to increase or decrease the number of devices to be reserved for manual configuration.

Old ISA based cards need to be manually configured which is explained here. You only need to read on here if you have one of these old cards. On embedded systems it might also be necessary to load the driver and then to configure the boards manually. In general manual configuration is done by running the **comedi_config** command (as `root`). Here is an example of how to use the command (perhaps you should read its **man** page now):

```
        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, and their meaning is driver dependant.

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 preceding **comedi_config** command to succeed, the `ni_atmio` kernel module must be loaded first. For plug-n-play boards on modern kernels, the appropriate comedi kernel modules should get loaded automatically when your computer is booted. The **modprobe** command can be used to manually load/unload kernel modules, and **lsmod** will list all the currently loaded modules.

For the Data Translation board, you need to know how the board's jumpers are configured in order to specify the correct **comedi_config** parameters. These parameters for the board are given in the kernel drivers section about the `dt282x` driver. The card discussed here is a DT2821-f-8di. The entry for the `dt282x` driver tells you that the **comedi_config** parameters give the driver the I/O base, IRQ, DMA 1, DMA 2, and in addition the states of the differential/single-ended and unipolar/bipolar jumpers:

DT282X 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]

So, the appropriate options list might be:

```
        0x200,4,0,0,1,1,1,1,0,2,2
```

and the full configuration command is:

```
        comedi_config /dev/comedi1 dt2821-f-8di 0x200,4,0,0,1,1,1,1,0,2,2
```

Setting the DMA channels to 0 disables the use of DMA.

So now you have your boards configured correctly. Since data acquisition boards are not typically well-engineered, Comedi sometimes can't figure out if an old non-plug-n-play board is actually in the computer and at the base address you specified. 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. Call the command **comedi_board_info**, which provides information about each subdevice on the board. Here's part of the output for the USB-DUX sigma board (which is on `/dev/comedi0`), as a result of the command **comedi_board_info -v**.

```
overall info:
  version code: 0x00074c
  driver name: usbduxsigma
  board name: usbduxsigma
  number of subdevices: 4
subdevice 0:
  type: 1 (analog input)
  flags: 0x10119000
          SDF_CMD_READ:can do asynchronous input commands
          SDF_READABLE:subdevice can be read
          SDF_GROUND:can do aref=ground
          SDF_LSAMPL:subdevice uses 32-bit samples for commands
  number of channels: 16
  max data value: 16777215
  ranges:
    all chans: [-1.325 V,1.325 V]
  command:
    start: now|int
    scan_begin: timer
    convert: now
    scan_end: count
    stop: none|count
  command structure filled with probe_cmd_generic_timed for 16 channels:
    start: now 0
    scan_begin: timer 1000000
      scan_begin_src = TRIG_TIMER:
      The sampling rate is defined per scan
      meaning all channels are sampled at
      the same time. The maximum sampling rate is f=1000 Hz
    convert: now 0
    scan_end: count 16
    stop: count 2
subdevice 1:
  type: 2 (analog output)
  flags: 0x00125000
          SDF_CMD_WRITE:can do asynchronous output commands
          SDF_WRITABLE:subdevice can be written
          SDF_GROUND:can do aref=ground
  number of channels: 4
  max data value: 255
  ranges:
    all chans: [0 V,2.5 V]
  command:
    start: now|int
    scan_begin: timer
    convert: now
    scan_end: count
    stop: none|count
  command structure filled with probe_cmd_generic_timed for 4 channels:
    start: now 0
    scan_begin: timer 1000000
      scan_begin_src = TRIG_TIMER:
      The sampling rate is defined per scan
      meaning all channels are sampled at
      the same time. The maximum sampling rate is f=1000 Hz
    convert: now 0
```

```
    scan_end: count 4
    stop: count 2
subdevice 2:
  type: 5 (digital I/O)
  flags: 0x00030000
         SDF_READABLE:subdevice can be read
         SDF_WRITABLE:subdevice can be written
  number of channels: 24
  max data value: 1
  ranges:
    all chans: [0 V,5 V]
  command:
    not supported
subdevice 3:
  type: 12 (pwm)
  flags: 0x00020100
         SDF_MODE1:can do mode 1
         SDF_WRITABLE:subdevice can be written
  number of channels: 8
  max data value: 512
  ranges:
    all chans: [0,1]
  command:
    not supported
```

This board has four subdevices. Devices are separated into subdevices that each have a distinct purpose; e.g., analog input, analog output, digital input/output.

Here's the information from Comedi's `/proc/comedi` file, which indicates what drivers are loaded and which boards are configured:

```
cat /proc/comedi
```

```
comedi version 0.7.76
format string: "%2d: %-20s %-20s %4d",i,driver_name,board_name,n_subdevices
 0: usbduxsigma          usbduxsigma             4
usbduxfast:
 usbduxfast
usbduxsigma:
 usbduxsigma
```

This documentation feature currently returns the driver name, the device name, and the number of subdevices. Following those lines are a list of the Comedi kernel driver modules currently loaded, each followed by a list of the board names it recognizes (names that can be used with **comedi_config**).

3 Writing Comedi programs

This section describes how Comedi 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:

```
 /*
 * Tutorial example #1
 * Part of Comedilib
 *
 * Copyright (c) 1999,2000 David A. Schleef <ds@schleef.org>
 *
 * This file may be freely modified, distributed, and combined with
 * other software, as long as proper attribution is given in the
 * source code.
 */

#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;
  int chan = 0;
  lsampl_t data;
  int retval;

  it = comedi_open("/dev/comedi0");
  if(it == NULL) {
    comedi_perror("comedi_open");
    return 1;
  }

  retval = comedi_data_read(it, subdev, chan, range, aref, &data);
  if(retval < 0) {
    comedi_perror("comedi_data_read");
    return 1;
  }

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

  return 0;
}
```

The source code file for the above program can be found in Comedilib, at `demo/tut1.c`. You can compile the program using

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

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

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 between integer data and physical units

If you selected an analog input subdevice, you probably noticed that the output of **tut1** is an unsigned number, for example between `0` and `65535` for a 16 bit analog input. Comedi samples are unsigned, with `0` representing the lowest voltage of the

ADC, and a hardware-dependent maximum value representing the highest voltage. Comedi compensates for anything else the manual for your device says (for example, many boards represent bipolar analog input voltages as signed integers). 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?"

The functions comedi_to_physical(), comedi_to_phys(), comedi_from_physical() and comedi_from_phys() are used to convert between Comedi's integer data and floating point numbers corresponding to physical values (voltages, etc.).

3.3 Your second Comedi program

Actually, this is the first Comedi program again, except we've added code to convert the integer data value to physical units.

```
      /*
 * Tutorial example #2
 * Part of Comedilib
 *
 * Copyright (c) 1999,2000 David A. Schleef <ds@schleef.org>
 * Copyright (c) 2008 Frank Mori Hess <fmhess@users.sourceforge.net>
 *
 * This file may be freely modified, distributed, and combined with
 * other software, as long as proper attribution is given in the
 * source code.
 */

#include <stdio.h>    /* for printf() */
#include <stdlib.h>
#include <comedilib.h>
#include <ctype.h>
#include <math.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 */
const char filename[] = "/dev/comedi0";

int main(int argc, char *argv[])
{
  comedi_t *device;
  lsampl_t data;
  double physical_value;
  int retval;
  comedi_range *range_info;
  lsampl_t maxdata;

  device = comedi_open(filename);
  if (device == NULL) {
    comedi_perror(filename);
    return 1;
  }

  retval = comedi_data_read(device, subdev, chan, range, aref,
          &data);
  if (retval < 0) {
    comedi_perror(filename);
    return 1;
  }

  comedi_set_global_oor_behavior(COMEDI_OOR_NAN);
  range_info = comedi_get_range(device, subdev, chan, range);
  maxdata = comedi_get_maxdata(device, subdev, chan);
```

```
  printf("[0,%d] -> [%g,%g]\n", maxdata,
         range_info->min, range_info->max);
  physical_value = comedi_to_phys(data, range_info, maxdata);
  if (isnan(physical_value)) {
    printf("Out of range [%g,%g]",
           range_info->min, range_info->max);
  } else {
    printf("%g", physical_value);
    switch(range_info->unit) {
    case UNIT_volt: printf(" V"); break;
    case UNIT_mA: printf(" mA"); break;
    case UNIT_none: break;
    default: printf(" (unknown unit %d)",
        range_info->unit);
    }
    printf(" (%lu in raw units)\n", (unsigned long)data);
  }
  return 0;
}
```

The source code file for the above program can be found in the Comedilib source at `demo/tut2.c` and if installed as a package usually at `/usr/share/doc/libcomedi-dev/demo/` with all the other tutorial/demo files.

3.4 Asynchronous acquisition

Of special importance is the so called "asynchronous data acquisition" where Comedi is sampling in the background at a given sample rate. The user can retrieve the data whenever it is convenient. Comedi stores the data in a ring-buffer so that programs can perform other tasks in the foreground, for example plotting data or interacting with the user. This technique is used in programs such as **ktimetrace** or **comedirecord**.

There are two different ways how a sequence of channels is measured during asynchronous acquisition (see also the Figure in the introduction):

- The channels are measured with the help of a multiplexer which switches to the next channel after each measurement. This means that the sampling rate is divided by the number of channels.
- The channels are all measured at the same time, for example when every channel has its own converter. In this case the sampling rate need not to be divided by the number of channels.

How your Comedi device handles the asynchronous acquisition can be found out with the command **comedi_board_info -v**.

The program `demo/tut3.c` demonstrates the asynchronous acquisition. The general strategy is always the same: first, we tell Comedi all sampling parameters such as the sampling rate, the number of channels and anything it needs to know so that it can run independently in the background. Then Comedi checks our request and it might modify it. For example we might want to have a sampling rate of 16kHz but we only get 1kHz. Finally we can start the asynchronous acquisition. Once it has been started we need to check periodically if data is available and request it from Comedi so that its internal buffer won't overrun.

In summary the asynchonous acquisition is performed in the following way:

- Create a command structure of type comedi_cmd
- Call the function `comedi_get_cmd_generic_timed`() to fill the command structure with your comedi device, subdevice, sampling rate and number of channels.
- Create a channel-list and store it in the command structure. This tells comedi which channels should be sampled in the background.
- Call `comedi_command_test`() with your command structure. Comedi might modify your requested sampling rate and channels.

- Call comedi_command_test() again which now should return zero for success.
- Call comedi_command() to start the asynchronous acquisition. From now on the kernel ringbuffer will be filled at the specified sampling rate.
- Call periodically the standard function read() and receive the data. The result should always be non zero as long as the acquisition is running.
- Convert the received data either into lsampl_t or sampl_t depending on the subdevice flag SDF_LSAMPL.
- Poll for data with read() as long as it returns a positive result or until the program terminates.

The program below is a stripped down version of the program cmd.c in the demo directory. It requests data from two channels at a sampling rate of 1kHz and a total of 10000 samples. which are then printed to stdout. You can pipe the data into a file and plot it with gnuplot. As mentioned above, central in this program is the loop using the standard C read() command which receives the buffer contents.

```
      /*
 * Example of using commands - asynchronous input
 * Part of Comedilib
 *
 * Copyright (c) 1999,2000,2001 David A. Schleef <ds@schleef.org>
 *               2008 Bernd Porr <berndporr@f2s.com>
 *
 * This file may be freely modified, distributed, and combined with
 * other software, as long as proper attribution is given in the
 * source code.
 */

#include <stdio.h>
#include <comedilib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/time.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>

extern comedi_t *device;

struct parsed_options {
  char *filename;
  double value;
  int subdevice;
  int channel;
  int aref;
  int range;
  int verbose;
  int n_chan;
  int n_scan;
  double freq;
};



#define BUFSZ 10000
char buf[BUFSZ];

#define N_CHANS 256
static unsigned int chanlist[N_CHANS];
static comedi_range * range_info[N_CHANS];
static lsampl_t maxdata[N_CHANS];
```

```
int prepare_cmd_lib(comedi_t *dev, int subdevice, int n_scan,
        int n_chan, unsigned period_nanosec,
        comedi_cmd *cmd);

void do_cmd(comedi_t *dev,comedi_cmd *cmd);

void print_datum(lsampl_t raw, int channel_index);

char *cmdtest_messages[] = {
  "success",
  "invalid source",
  "source conflict",
  "invalid argument",
  "argument conflict",
  "invalid chanlist",
};

int main(int argc, char *argv[])
{
  comedi_t *dev;
  comedi_cmd c,*cmd = &c;
  int ret;
  int total = 0;
  int col;
  int i;
  int subdev_flags;
  lsampl_t raw;

  struct parsed_options options;

  memset(&options, 0, sizeof(options));
  options.filename = "/dev/comedi0";
  options.subdevice = 0;
  options.channel = 0;
  options.range = 0;
  options.aref = AREF_GROUND;
  options.n_chan = 2;
  options.n_scan = 10000;
  options.freq = 1000.0;

  /* open the device */
  dev = comedi_open(options.filename);
  if (!dev) {
    comedi_perror(options.filename);
    exit(1);
  }

  /* Print numbers for clipped inputs */
  comedi_set_global_oor_behavior(COMEDI_OOR_NUMBER);

  /* Set up channel list */
  for (i = 0; i < options.n_chan; i++) {
    chanlist[i] =
      CR_PACK(options.channel + i, options.range,
        options.aref);
    range_info[i] =
      comedi_get_range(dev, options.subdevice,
           options.channel, options.range);
    maxdata[i] =
      comedi_get_maxdata(dev, options.subdevice,
```

```
            options.channel);
}

/* prepare_cmd_lib() uses a Comedilib routine to find a
 * good command for the device.  prepare_cmd() explicitly
 * creates a command, which may not work for your device. */
prepare_cmd_lib(dev, options.subdevice, options.n_scan,
    options.n_chan, 1e9 / options.freq, cmd);

/* comedi_command_test() tests a command to see if the
 * trigger sources and arguments are valid for the subdevice.
 * If a trigger source is invalid, it will be logically ANDed
 * with valid values (trigger sources are actually bitmasks),
 * which may or may not result in a valid trigger source.
 * If an argument is invalid, it will be adjusted to the
 * nearest valid value.  In this way, for many commands, you
 * can test it multiple times until it passes.  Typically,
 * if you can't get a valid command in two tests, the original
 * command wasn't specified very well. */
ret = comedi_command_test(dev, cmd);
if (ret < 0) {
  comedi_perror("comedi_command_test");
  if(errno == EIO){
    fprintf(stderr,
      "Ummm... this subdevice doesn't support commands\n");
  }
  exit(1);
}
ret = comedi_command_test(dev, cmd);
if (ret < 0) {
  comedi_perror("comedi_command_test");
  exit(1);
}
fprintf(stderr,"second test returned %d (%s)\n", ret,
  cmdtest_messages[ret]);
if (ret != 0) {
  fprintf(stderr, "Error preparing command\n");
  exit(1);
}

/* comedi_set_read_subdevice() attempts to change the current
 * 'read' subdevice to the specified subdevice if it is
 * different.  Changing the read or write subdevice might not be
 * supported by the version of Comedi you are using.  */
comedi_set_read_subdevice(dev, cmd->subdev);
/* comedi_get_read_subdevice() gets the current 'read'
 * subdevice. if any.  This is the subdevice whose buffer the
 * read() call will read from.  Check that it is the one we want
 * to use.  */
ret = comedi_get_read_subdevice(dev);
if (ret < 0 || ret != cmd->subdev) {
  fprintf(stderr,
    "failed to change 'read' subdevice from %d to %d\n",
    ret, cmd->subdev);
  exit(1);
}

/* start the command */
ret = comedi_command(dev, cmd);
if (ret < 0) {
  comedi_perror("comedi_command");
  exit(1);
```

```
  }
  subdev_flags = comedi_get_subdevice_flags(dev, options.subdevice);
  col = 0;
  while (1) {
    ret = read(comedi_fileno(dev),buf,BUFSZ);
    if (ret < 0) {
      /* some error occurred */
      perror("read");
      break;
    } else if (ret == 0) {
      /* reached stop condition */
      break;
    } else {
      int bytes_per_sample;

      total += ret;
      if (options.verbose) {
        fprintf(stderr, "read %d %d\n", ret,
          total);
      }
      if (subdev_flags & SDF_LSAMPL) {
        bytes_per_sample = sizeof(lsampl_t);
      } else {
        bytes_per_sample = sizeof(sampl_t);
      }
      for (i = 0; i < ret / bytes_per_sample; i++) {
        if (subdev_flags & SDF_LSAMPL) {
          raw = ((lsampl_t *)buf)[i];
        } else {
          raw = ((sampl_t *)buf)[i];
        }
        print_datum(raw, col);
        col++;
        if (col == options.n_chan) {
          printf("\n");
          col=0;
        }
      }
    }
  }

  return 0;
}

/*
 * This prepares a command in a pretty generic way.  We ask the
 * library to create a stock command that supports periodic
 * sampling of data, then modify the parts we want.
 */
int prepare_cmd_lib(comedi_t *dev, int subdevice, int n_scan,
        int n_chan, unsigned scan_period_nanosec,
        comedi_cmd *cmd)
{
  int ret;

  memset(cmd,0,sizeof(*cmd));

  /* This comedilib function will get us a generic timed
   * command for a particular board.  If it returns -1,
   * that's bad. */
  ret = comedi_get_cmd_generic_timed(dev, subdevice, cmd, n_chan,
              scan_period_nanosec);
```

```
  if (ret < 0) {
    fprintf(stderr,
      "comedi_get_cmd_generic_timed failed\n");
    return ret;
  }

  /* Modify parts of the command */
  cmd->chanlist = chanlist;
  cmd->chanlist_len = n_chan;
  if (cmd->stop_src == TRIG_COUNT) {
    cmd->stop_arg = n_scan;
  }

  return 0;
}

void print_datum(lsampl_t raw, int channel_index)
{
  double physical_value;

  physical_value = comedi_to_phys(raw, range_info[channel_index],
          maxdata[channel_index]);
  printf("%#8.6g ", physical_value);
}
```

The source code file for the above program can be found in Comedilib, at `demo/tut3.c`. You can compile the program using

```
      cc tut3.c -lcomedi -lm -o tut3
```

For advanced programmers the function `comedi_get_buffer_contents`() is useful to check if there is actually data in the ringbuffer so that a call of `read`() can be avoided for example when the data readout is called by a timer call-back function.

3.5 Further examples

See the `demo` subdirectory of Comedilib for more example programs. The directory contains a `README` file with descriptions of the various demo programs.

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 5 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 comedi_dio_read() and comedi_dio_write():

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

int **comedi_dio_write**(comedi_t *device, unsigned int subdevice, unsigned int 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():

int **comedi_dio_config**(comedi_t *device, unsigned int subdevice, unsigned int 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_bitfield2():

int **comedi_dio_bitfield2**(comedi_t *device, unsigned int subdevice, unsigned int write_mask, unsigned int *bits, unsigned int base_channel);

Each channel from *base_channel* to *base_channel* + 31 is assigned to a bit in the *write_mask* and *bits* bitfield with bit 0 assigned to channel *base_channel*, bit 1 assigned to channel *base_channel* + 1, etc. If a bit in *write_mask* is set, the corresponding bit in **bits* will be written to the digital output line corresponding to the channel given by *base_channel* plus the bit number. 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 *base_channel* + 0 is the least significant bit in the bitfield. No more than 32 channels at once can be accessed using this method. **Warning!** Older versions of Comedi may ignore *base_channel* and treat it as 0 unless the subdevice has more than 32 channels.

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.

Single samples can be read from an analog channel using the function comedi_data_read():

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

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

The *range* parameter is the zero-based index of one of the gain ranges supported by the channel. This is a number from 0 to N-1 where N is the number of ranges supported by the channel. Use the function comedi_get_n_ranges() to get the number of ranges supported by the channel, the function comedi_find_range() to search for a suitable range, or the function comedi_get_range() to get the details of a supported range.

The *aref* parameter specifies an analog reference to use: AREF_GROUND, AREF_COMMON, AREF_DIFF, or AREF_OTHER. Use the function comedi_get_subdevice_flags() to see which analog references are supported by the subdevice.

In the opposite direction, single samples can be written to an analog output channel using the function comedi_data_write():

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

Raw data values read or written by the above functions 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():

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

Conversion between raw data values and uncalibrated physical units can be performed by the functions comedi_to_phys() and comedi_from_phys():

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

lsampl_t **comedi_from_phys**(double data, comedi_range *range, lsampl_t maxdata);

There are some 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".
- comedi_range: this holds the minimum and maximum physical values for a gain range supported by a channel of a subdevice, and specifies the units. This can be used in combination with the channel's "maxdata" value to convert between unsigned integer sample values (of type lsampl_t or sampl_t) and physical units in a nominal (uncalibrated) way using the comedi_to_phys() and comedi_from_phys() functions. Use the comedi_get_maxdata() function to get the "maxdata" value for the channel.

  Most functions specify the range to be used for a channel by a zero-based index into the list of ranges supported by the channel. Depending on the device and subdevice, different channels on the subdevice may or may not share the same list of ranges, that is, ranges may or may not be channel-specific. (The SDF_RANGETYPE subdevice flag indicates whether ranges are channel-specific.)

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, comedi_data_read_n():

int **comedi_data_read_n**(comedi_t *device, unsigned int subdevice, unsigned int channel, 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 a single data acquisition can also be delayed by a specified number of nano-seconds using the function comedi_data_read_delayed():

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 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:

```
typedef struct comedi_insn_struct {
  unsigned int insn;      // integer encoding the type of acquisition
                          // (or configuration)
  unsigned int n;         // number of elements in data array
  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* member 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 with the function comedi_do_insn():

int **comedi_do_insn**(comedi_t *device, 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.

A list of instructions can be executed in one function call using the function comedi_do_insnlist():

int **comedi_do_insnlist**(comedi_t *device, comedi_insnlist *list);

The parameter *list* is a pointer to a comedi_insnlist data structure holding a pointer to an array of comedi_insn and the number of instructions in the list:

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

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 subdevice. 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* member in an instruction data structure indicates that the instruction will *not perform acquisition* on a channel, but will *configure* that channel. The *chanspec* member in the comedi_insn data structure, contains the channel to be configured. The zeroth element of the data array is always an id that specifies what type of configuration instruction is being performed. The meaning of rest of the elements in the data array depend on the configuration instruction id. Some of the possible ids are summarised in the table below, along with the meanings of the data array elements for each type of configuration instruction.

data[0]	Description	n (number of elements in data array)	Meanings of data[1], ..., data[n-1]
INSN_CONFIG_DIO_INPUT	Configure a DIO line as input. It is easier to use comedi_dio_config() than to use this configuration instruction directly.	1	n/a
INSN_CONFIG_DIO_OUTPUT	Configure a DIO line as output. It is easier to use comedi_dio_config() than to use this configuration instruction directly.	1	n/a
INSN_CONFIG_ALT_SOURCE	Select an alternate input source. This instruction is used by calibration programs to configure analog input channels which can be redirected to read internal calibration references. You need to set the CR_ALT_SOURCE flag in the chanspec when reading to actually read from the configured alternate input source. If you are using comedi_data_read(), then the channel parameter can be bitwise or'd with the CR_ALT_SOURCE flag.	2	data[1]: alternate input source.
INSN_CONFIG_BLOCK_SIZE	Specify block size for asynchonous command data. When performing streaming input, many boards accumulate samples in internal fifos and transfer them to the host computer in chunks. Some drivers let you suggest a size in bytes for how big a the chunks should be. This lets you tune how often the host computer is interrupted with a new chunk of data.	2	data[1]: The desired block size in bytes. The actual configured block size is writen back to data[1] after the instruction completes. This instruction acts purely as a query if the block size is set to zero.
INSN_CONFIG_DIO_QUERY	Queries the configuration of a DIO line to see if it is an input or output. It is probably easier to use the comedilib function comedi_dio_get_config() than to use this instruction directly.	2	data[1]: The instruction sets this element to either COMEDI_INPUT or COMEDI_OUTPUT.

See the comedilib demo program demo/choose_clock.c for an example of using a configuration instruction.

4.4 Instruction for internal triggering

This special instruction has INSN_INTTRIG as the *insn* member 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] element 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 command is executed by the function comedi_command():

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 specified 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 Section 4.5.4.

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" event occurs *start_arg* nanoseconds after the command is set up. Currently, only *start_arg*=0 is supported.
- TRIG_FOLLOW: (For an output device.) The "start" 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_src* events. The available options are:

- TRIG_TIMER: "scan begin" events occur periodically. The time between "scan begin" events is *scan_begin_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_src* events. The available options are:

- 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 by setting the *scan_end_src* event to 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* event. The available options are:

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

  stop_arg is used to denote how many samples should be used in the continuous acquisition. If *stop_arg* is set to 0, the entire output buffer may contribute to the output. If *stop_arg* != 0, only the memory for *stop_arg* samples will be used. Many drivers do not yet support *stop_arg*!=0 and should enforce *stop_arg*=0 via comedi_command_test.

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 or RTLinux/GPL) 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

Note: The functions described here use an old feature that is no longer implemented by the Comedi kernel layer. THEY WILL NOT WORK!

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 channel);

The above function `comedi_sv_init`() initializes the comedi_sv_t data structure, used to do the averaging acquisition:

```
typedef 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;
} comedi_sv_t;
```

The actual acquisition is done with the function `comedi_sv_measure`():

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

The number of samples over which the function `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 subdevice, the command triggering portion of the flags are ignored.

data[2] determines the mode of operation. The mode of operation is actually a bitfield that encodes what to do for various transitions of the source signals.

data[3], data[4] determine the primary source for the counter, similar to the *..._src* and the *..._arg* fields used in the command data structure.

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

4.7.9 National Instruments General Purpose Counters/Timers (GPCT)

Counters/timers and pulse generators are fairly different in terms of functionality, but they correspond to similar devices seen either as input or output. When generalising, these devices are both referred to as "counters". The NI boards provide a couple of such counters, under the name of GPCT. A counter is made of the following basic elements:

Input source the signal measured or the clock of the pulse generation.

Gate controls when the counting (or sampling) occurs.

Register holds the current count.

Out for the output counters (pulse generators), the output signal.

There are many different ways to count, time or generate pulses. All the modes rely on the counter and some particular configuration. For example, in a typical buffered counting mode, the source is the (digital) signal that is measured, the counter is increased at every rising edge of the signal, the gate is the (digital) signal indicating when to save the counter to memory. It is preferable you get first familiarized with these various modes by reading your NI board documentation before reading the following description on the mapping to the comedi interface.

Each counter of the board is represented in comedi as a subdevice of type COMEDI_SUBD_COUNTER. Each subdevice has a device file associated (eg, /dev/comedi0_subd11) in order to read or write buffered data from or to the counter. Note that the comedi subdevice has three "channels". In most case, only channel 0 is to be used. Reading or writing on channel 0 corresponds to reading/writing the counter value. The GPCT also has two registers named A and B, they can be accessed respectively via channels 1 and 2.

To configure the behaviour of the counter with comedi, the function comedi_set_counter_mode() is used. The possible mode values are to be found in the ni_gpct_mode_bits enum constants. For instance, by default the counter is cumulative, even in buffered counting. To reinitialise it after each sampling (ie, after an edge on the gate signal), one can add the NI_GPCT_LOADING_ON_GATE_BIT to the mode. In that case, the counter will be reset to the value of the A register after each sampling.

To configure the signal to be used as the "source", one uses the comedi_set_clock_source() with one constant from the ni_gpct_clock_source_bits enum. When the period of the signal is fixed and known, it should be specified as the last parameter of the method, otherwise 0 should be passed. Note that in comedi this is called "clock" because in timer and pulse generator, this signal is used as the clock.

To configure the signal to be used as the "gate", one uses the comedi_set_gate_source() with one constant from the ni_gpct_gate_select enum. When the gate signal is not be used, NI_GPCT_DISABLED_GATE_SELECT should be specified. Some NI boards have two gates, but the behaviour associated with the second gate is usually unknown so it is recommended to disable it. Note that this is called "gate" because in some modes, this signal is used to block/unblock the counter.

The function comedi_reset() will stop and reset a counter. After being configured, to start a counter, it should be "armed", which can be done either with the comedi_arm() function (for simple counting mode), or with the *start_src* member of the command (for buffered counting).

One side thing to mention is the signal routing of the NI card, which is done via the PFIs (Programmable Function Inputs). NI's naming is confusing because they use the same name for the terminal (ie, physical input/output pins) and for the signal (ie, the logical information that controls/indicates a specific event). The routing allows to configure which signal goes to a PFI terminal. This is done via comedi_set_routing(), with subdevice being the special DIO comedi subdevice (eg, 7 on M-series), the PFI terminal number as channel, the signal that should be routed to it encoded as source with one of the constants from the ni_pfi_routing enum. The direction of the pin must also be correctly configured (ie, whether it is used as input or output). This is done via comedi_dio_config() with the same subdevice and channel, and either COMEDI_INPUT or COMEDI_OUTPUT.

4.7.10 National Instruments RTSI trigger bus

A number of NI boards support the RTSI (Real Time System Integration) bus. It's primary use is to synchronize multiple DAQ cards. On PXI boards, the RTSI lines correspond to the PXI trigger lines 0 to 7. PCI boards use cables to connect to their RTSI ports. The RTSI bus consists of 8 digital signal lines numbered 0 to 7 that are bi-directional. Each of these signal lines can be configured as an input or output, and the signal appearing on the output of each line can be configured to one of several internal board timing signals (although on older boards RTSI line 7 can only be used for the clock signal). The ni_pcimio, ni_atmio, and ni_mio_cs drivers expose the RTSI bus as a digital I/O subdevice (subdevice number 10).

The functions comedi_dio_config() and comedi_dio_get_config() can be used on the RTSI subdevice to set/query the direction (input or output) of each of the RTSI lines individually.

The subdevice also supports the INSN_CONFIG_SET_CLOCK_SRC and INSN_CONFIG_GET_CLOCK_SRC configuration instructions, which can be used to configure/query what source the board uses to synchronize its master clock to. The various possibilities are defined in the comedi.h header file:

Clock Source	Description
NI_MIO_INTERNAL_CLOCK	Use the board's internal oscillator.
NI_MIO_RTSI_CLOCK	Use the RTSI line 7 as the master clock. This source is only supported on pre-m-series boards. The newer m-series boards use NI_MIO_PLL_RTSI_CLOCK() instead.
NI_MIO_PLL_PXI_STAR_TRIGGER_CLOCK	Only available for newer m-series PXI boards. Synchronizes the board's phased-locked loop (which runs at 80MHz) to the PXI star trigger line.
NI_MIO_PLL_PXI10_CLOCK	Only available for newer m-series PXI boards. Synchronizes the board's phased-locked loop (which runs at 80MHz) to the 10 MHz PXI backplane clock.
NI_MIO_PLL_RTSI_CLOCK(*n*)	Only available for newer m-series boards. The function returns a clock source which will cause the board's phased-locked loop (which runs at 80MHz) to syncronize to the RTSI line specified in the function argument.

For all clock sources except NI_MIO_INTERNAL_CLOCK and NI_MIO_PLL_PXI10_CLOCK, you should pass the period of the clock your are feeding to the board when using INSN_CONFIG_SET_CLOCK_SRC.

Finally, the configuration instructions INSN_CONFIG_SET_ROUTING and INSN_CONFIG_GET_ROUTING can be used to select/query which internal signal will appear on a given RTSI output line. The header file comedi.h defines the following signal sources which can be routed to an RTSI line:

Signal Source	Description
NI_RTSI_OUTPUT_ADR_START1	ADR_START1, an analog input start signal. See the NI's DAQ-STC Technical Reference Manual for more information.
NI_RTSI_OUTPUT_ADR_START2	ADR_START2, an analog input stop signal. See the NI's DAQ-STC Technical Reference Manual for more information.
NI_RTSI_OUTPUT_SCLKG	SCLKG, a sample clock signal. See the NI's DAQ-STC Technical Reference Manual for more information.
NI_RTSI_OUTPUT_DACUPDN	DACUPDN, a dac update signal. See the NI's DAQ-STC Technical Reference Manual for more information.
NI_RTSI_OUTPUT_DA_START1	DA_START1, an analog output start signal. See the NI's DAQ-STC Technical Reference Manual for more information.
NI_RTSI_OUTPUT_G_SRC0	G_SRC0, the source signal to general purpose counter 0. See the NI's DAQ-STC Technical Reference Manual for more information.
NI_RTSI_OUTPUT_G_GATE0	G_GATE0, the gate signal to general purpose counter 0. See the NI's DAQ-STC Technical Reference Manual for more information.
NI_RTSI_OUTPUT_RGOUT0	RGOUT0, the output signal of general purpose counter 0. See the NI's DAQ-STC Technical Reference Manual for more information.
NI_RTSI_OUTPUT_RTSI_BRD(n)	RTSI_BRD0 though RTSI_BRD3 are four internal signals which can have various other signals routed to them in turn. Currently, comedi provides no way to configure the signals routed to the RTSI_BRD lines. See the NI's DAQ-STC Technical Reference Manual for more information.
NI_RTSI_OUTPUT_RTSI_OSC	The RTSI clock signal. On pre-m-series boards, this signal is always routed to RTSI line 7, and cannot be routed to lines 0 through 6. On m-series boards, any RTSI line can be configured to output the clock signal.

The RTSI bus pins may be used as trigger inputs for many of the Comedi trigger functions. To use the RTSI bus pins, set the source to be TRIG_EXT and the source argument using the return values from the NI_EXT_RTSI(n) function (or similarly the NI_EXT_PFI(n) function if you want to trigger from a PFI line). The CR_EDGE and CR_INVERT flags may also be set on the trigger source argument to specify edge and falling edge/low level triggering.

An example to set up a device as a master is given below.

```
void comediEnableMaster(comedi_t *dev){
        comedi_insn   configCmd;
        lsampl_t      configData[2];
        int           ret;
        unsigned int  d = 0;
        static const unsigned rtsi_subdev = 10;
        static const unsigned rtsi_clock_line = 7;

        /* Route RTSI clock to line 7 (not needed on pre-m-series boards since their
           clock is always on line 7). */
        memset(&configCmd, 0, sizeof(configCmd));
        memset(&configData, 0, sizeof(configData));
```

```
        configCmd.insn = INSN_CONFIG;
        configCmd.subdev = rtsi_subdev;
        configCmd.chanspec = rtsi_clock_line;
        configCmd.n = 2;
        configCmd.data = configData;
        configCmd.data[0] = INSN_CONFIG_SET_ROUTING;
        configCmd.data[1] = NI_RTSI_OUTPUT_RTSI_OSC;
        ret = comedi_do_insn(dev, &configCmd);
        if(ret < 0){
                comedi_perror("comedi_do_insn: INSN_CONFIG");
                exit(1);
        }
        // Set clock RTSI line as output
        ret = comedi_dio_config(dev, rtsi_subdev, rtsi_clock_line, INSN_CONFIG_DIO_OUTPUT);
        if(ret < 0){
                comedi_perror("comedi_dio_config");
                exit(1);
        }

        /* Set routing of the 3 main AI RTSI signals and their direction to output.
           We're reusing the already initialized configCmd instruction here since
           it's mostly the same. */
        configCmd.chanspec = 0;
        configCmd.data[1] =  NI_RTSI_OUTPUT_ADR_START1;
        ret = comedi_do_insn(dev, &configCmd);
        if(ret < 0){
                comedi_perror("comedi_do_insn: INSN_CONFIG");
                exit(1);
        }
        ret = comedi_dio_config(dev, rtsi_subdev, 0, INSN_CONFIG_DIO_OUTPUT);
        if(ret < 0){
                comedi_perror("comedi_dio_config");
                exit(1);
        }

        configCmd.chanspec = 1;
        configCmd.data[1] =  NI_RTSI_OUTPUT_ADR_START2;
        ret = comedi_do_insn(dev, &configCmd);
        if(ret < 0){
                comedi_perror("comedi_do_insn: INSN_CONFIG");
                exit(1);
        }
        ret = comedi_dio_config(dev, rtsi_subdev, 1, INSN_CONFIG_DIO_OUTPUT);
        if(ret < 0){
                comedi_perror("comedi_dio_config");
                exit(1);
        }

        configCmd.chanspec = 2;
        configCmd.data[1] =  NI_RTSI_OUTPUT_SCLKG;
        ret = comedi_do_insn(dev, &configCmd);
        if(ret < 0){
                comedi_perror("comedi_do_insn: INSN_CONFIG");
                exit(1);
        }
        ret = comedi_dio_config(dev, rtsi_subdev, 2, INSN_CONFIG_DIO_OUTPUT);
        if(ret < 0){
                comedi_perror("comedi_dio_config");
                exit(1);
        }
}
```

An example to slave a m-series device from this master follows. A pre-m-series device would need to use NI_MIO_RTSI_CLOCK for the clock source instead. In your code, you may also wish to configure the master device to use the external clock source instead of using its internal clock directly (for best syncronization).

```
void comediEnableSlave(comedi_t *dev){
        comedi_insn   configCmd;
        lsampl_t      configData[3];
        int           ret;
        unsigned int  d = 0;;
        static const unsigned rtsi_subdev = 10;
        static const unsigned rtsi_clock_line = 7;

        memset(&configCmd, 0, sizeof(configCmd));
        memset(&configData, 0, sizeof(configData));
        configCmd.insn = INSN_CONFIG;
        configCmd.subdev = rtsi_subdev;
        configCmd.chanspec = 0;
        configCmd.n = 3;
        configCmd.data = configData;
        configCmd.data[0] = INSN_CONFIG_SET_CLOCK_SRC;
        configCmd.data[1] = NI_MIO_PLL_RTSI_CLOCK(rtsi_clock_line);
        configCmd.data[2] = 100;        /* need to give it correct external clock period */
        ret = comedi_do_insn(dev, &configCmd);
        if(ret < 0){
                comedi_perror("comedi_do_insn: INSN_CONFIG");
                exit(1);
        }
        /* configure RTSI clock line as input */
        ret = comedi_dio_config(dev, rtsi_subdev, rtsi_clock_line, INSN_CONFIG_DIO_INPUT);
        if(ret < 0){
                comedi_perror("comedi_dio_config");
                exit(1);
        }
        /* Configure RTSI lines we are using for AI signals as inputs. */
        ret = comedi_dio_config(dev, rtsi_subdev, 0, INSN_CONFIG_DIO_INPUT);
        if(ret < 0){
                comedi_perror("comedi_dio_config");
                exit(1);
        }
        ret = comedi_dio_config(dev, rtsi_subdev, 1, INSN_CONFIG_DIO_INPUT);
        if(ret < 0){
                comedi_perror("comedi_dio_config");
                exit(1);
        }
        ret = comedi_dio_config(dev, rtsi_subdev, 2, INSN_CONFIG_DIO_INPUT);
        if(ret < 0){
                comedi_perror("comedi_dio_config");
                exit(1);
        }
}

int comediSlaveStart(comedi_t *dev){
        comedi_cmd     cmd;
        unsigned int   nChannels = 8;
        double         sampleRate = 50000;
        unsigned int   chanList[8];
        int            i;

        // Setup chan list
        for(i = 0; i < nChannels; i++){
                chanList[i] = CR_PACK(i, 0, AREF_GROUND);
        }
```

```
        // Set up command
        memset(&cmd, 0, sizeof(cmd));
        ret = comedi_get_cmd_generic_timed(dev, subdevice, &cmd,
                (int)(1e9/(nChannels * sampleRate)));
        if(ret<0){
                printf("comedi_get_cmd_generic_timed failed\n");
                return ret;
        }
        cmd.chanlist        = chanList;
        cmd.chanlist_len    = nChannels;
        cmd.scan_end_arg    = nChannels;
        cmd.start_src        = TRIG_EXT;
        cmd.start_arg        = CR_EDGE | NI_EXT_RTSI(0);
        cmd.convert_src    = TRIG_EXT;
        cmd.convert_arg    = CR_INVERT | CR_EDGE | NI_EXT_RTSI(2);
        cmd.stop_src        = TRIG_NONE;

        ret = comedi_command(dev0, &cmd0);
        if(ret<0){
                printf("comedi_command failed\n");
                return ret;
        }
        return 0;
}
```

5 Comedi reference

5.1 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.

5.2 Constants and macros

5.2.1 CR_PACK

`CR_PACK(`*`chan`*`, `*`rng`*`, `*`aref`*`)` 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.

5.2.2 CR_PACK_FLAGS

CR_PACK_FLAGS(*chan*, *range*, *aref*, *flags*) is similar to CR_PACK() but can be used to combine one or more flag bits (bitwise-ORed together in the *flags* parameter) with the other parameters.

```
#define CR_PACK_FLAGS(chan, range, aref, flags) \
        (CR_PACK(chan, range, aref) | ((flags) & CR_FLAGS_MASK))
```

Depending on context, the *chan* parameter might not be a channel; it could be a trigger source, clock source, gate source etc. (in which case, the *range* and *aref* parameters would probably be set to 0), and the flags would modify the source in some device-dependant way.

The following flag values are defined:

CR_ALT_FILTER, CR_DITHER, CR_DEGLITCH (all the same) specify that some sort of filtering is to be done on the channel, trigger source, etc.

CR_ALT_SOURCE specifies that some alternate source is to be used for the channel (usually a calibration source).

CR_EDGE is usually combined with a trigger source number to specify that the trigger source is edge-triggered if the hardware and driver supports both edge-triggering and level-triggering. If both are supported, not asserting this flag specifies level-triggering.

CR_INVERT specifies that the trigger source, gate source, etc. is to be inverted.

5.2.3 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(*rangetype*) 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.

5.2.4 enum comedi_conversion_direction

```
enum comedi_conversion_direction
{
  COMEDI_TO_PHYSICAL,
  COMEDI_FROM_PHYSICAL
};
```

A comedi_conversion_direction is used to choose between converting data from Comedi's integer sample values to a physical value (COMEDI_TO_PHYSICAL), and converting from a physical value to Comedi's integer sample values (COMEDI_FROM_PHYSICAL).

5.2.5 enum comedi_io_direction

```
enum comedi_io_direction
{
  COMEDI_INPUT,
  COMEDI_OUTPUT
};
```

A comedi_io_direction is used to select between input or output. For example, comedi_dio_config() uses the COMEDI_INPUT and COMEDI_OUTPUT values to specify whether a configurable digital i/o channel should be configured as an input or output.

5.2.6 enum comedi_subdevice_type

```
enum comedi_subdevice_type {
  COMEDI_SUBD_UNUSED, /* subdevice is unused by driver */
  COMEDI_SUBD_AI, /* analog input */
  COMEDI_SUBD_AO, /* analog output */
  COMEDI_SUBD_DI, /* digital input */
  COMEDI_SUBD_DO, /* digital output */
  COMEDI_SUBD_DIO,  /* digital input/output */
  COMEDI_SUBD_COUNTER,  /* counter */
  COMEDI_SUBD_TIMER,  /* timer */
  COMEDI_SUBD_MEMORY, /* memory, EEPROM, DPRAM */
  COMEDI_SUBD_CALIB,  /* calibration DACs and pots*/
  COMEDI_SUBD_PROC, /* processor, DSP */
  COMEDI_SUBD_SERIAL, /* serial IO */
  COMEDI_SUBD_PWM /* pulse width modulation */
};
```

The comedi_subdevice_type enumeration specifies the possible values for a subdevice type. These values are used by the functions comedi_get_subdevice_type() and comedi_find_subdevice_by_type().

5.3 Data types and structures

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

```
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 described in Section 6.2.1.

5.3.1 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;  // index of subdevice whose buffer is read by read(), etc.  ↩
      on file descriptor from comedi_fileno() (negative means none)
  int          write_subdevice; // index of subdevice whose buffer is written by write(),  ↩
      etc. on file descriptor from comedi_fileno() (negatove means none).
  int          unused[30];
};
```

5.3.2 comedi_t

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

```
typedef struct comedi_t_struct comedi_t;
```

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 an opaque type, and pointers to type *comedi_t* should not be dereferenced by the application.

5.3.3 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 it is a 16-bit, unsigned integer.

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

5.3.4 lsampl_t

```
typedef unsigned int lsampl_t;
```

The data type lsampl_t is the data type typically used to represent data values in Comedilib. On most architectures it is a 32-bit, unsigned integer.

5.3.5 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 is no longer implemented by the Comedi kernel layer.

5.3.6 comedi_sv_t (deprecated)

```
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. This relies on the `COMEDI_TRIG` ioctl and is no longer by the Comedi kernel layer.

5.3.7 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.

5.3.8 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()` except the seconds and microseconds values are unsigned values of type lsampl_t.

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()`.

5.3.9 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.

5.3.10 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.

5.3.11 comedi_insnlist

```
typedef struct comedi_insnlist_struct comedi_insnlist;

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

A comedi_insnlist structure is used to communicate a list of instructions to the driver using the comedi_do_insnlist() function.

5.3.12 comedi_polynomial_t

```
#define COMEDI_MAX_NUM_POLYNOMIAL_COEFFICIENTS 4
typedef struct {
  double coefficients[COMEDI_MAX_NUM_POLYNOMIAL_COEFFICIENTS];
  double expansion_origin;
  unsigned order;
} comedi_polynomial_t;
```

A comedi_polynomial_t holds calibration data for a channel of a subdevice. It is initialized by the comedi_get_hardcal_converter() or comedi_get_softcal_converter() calibration functions and is passed to the comedi_to_physical() and comedi_from_physical() raw/physical conversion functions.

5.4 Functions

5.4.1 Core Functions

5.4.1.1 comedi_close

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 successful, comedi_close() returns 0. On failure, -1 is returned.

5.4.1.2 comedi_data_read

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 stores the value in **data*.

WARNING: comedi_data_read() does not do any pausing to allow multiplexed analog inputs to settle before starting 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().

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 functions comedi_to_phys() (linear conversion) or comedi_to_physical() (non-linear polynomial conversion).

Return value

On success, comedi_data_read() returns 1 (the number of samples read). If there is an error, -1 is returned.

5.4.1.3 comedi_data_read_n

comedi_data_read_n — read multiple samples from channel

Synopsis

```
#include <comedilib.h>
```

int **comedi_data_read_n**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int range, unsigned int aref, lsampl_t * data, unsigned int n);

Description

Similar to comedi_data_read() except it reads *n* samples into the array *data*. The precise timing of the samples is not hardware controlled.

5.4.1.4 comedi_data_read_delayed

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.

5.4.1.5 comedi_data_read_hint

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 inaccuracies 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.

5.4.1.6 comedi_data_write

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.

Return value

On success, comedi_data_write() returns 1 (the number of samples written). If there is an error, −1 is returned.

5.4.1.7 comedi_do_insn

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.

Return value

If successful, returns a non-negative number. For the case of `INSN_READ` or `INSN_WRITE` instructions, `comedi_do_insn`() returns the number of samples read or written, which may be less than the number requested. If there is an error, `-1` is returned.

5.4.1.8 comedi_do_insnlist

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. This function can be used to avoid the overhead of multiple system calls.

Return value

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

5.4.1.9 comedi_fileno

comedi_fileno — get file descriptor for open Comedilib device

Synopsis

```
#include <comedilib.h>
```

int **comedi_fileno**(comedi_t * device);

Description

The function comedi_fileno() returns the file descriptor for the device *device*. 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().

The returned file descriptor should not be closed, and will become invalid when comedi_close() is called on *device*.

Return value

A file descriptor, or -1 on error.

5.4.1.10 comedi_find_range

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 *channel* belonging to subdevice *subdevice* of the comedi device *device*, that includes both *min* and *max* in units of *unit*.

Return value

If a matching range is found, the index of the matching range is returned. If no matching range is available, the function returns -1.

5.4.1.11 comedi_find_subdevice_by_type

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*. The comedi_subdevice_type enum specifies the possible subdevice types.

Return value

If it finds a subdevice with the requested type, it returns its index. If there is an error, the function returns -1 and sets the appropriate error.

5.4.1.12 comedi_from_phys

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 parameter *data* given in physical units (double) into sample values (lsampl_t, between 0 and maxdata). 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 will be written to. The mapping between physical units and raw data is linear and assumes that the converter has ideal characteristics.

Conversion is not affected by out-of-range behavior. Out-of-range data parameters are silently truncated to the range 0 to *maxdata*.

5.4.1.13 comedi_from_physical

comedi_from_physical — convert physical units to sample using calibration data

Synopsis

```
#include <comedilib.h>
```

lsampl_t **comedi_from_physical**(double data, const comedi_polynomial_t * conversion_polynomial);

Description

Converts *data* given in physical units into Comedi's integer sample values (lsampl_t, between 0 and maxdata — see comedi_get_maxdata()). The *conversion_polynomial* parameter is obtained from either comedi_get_hardcal_converter() or comedi_get_softcal_converter(). The allows non linear and board specific correction. The result will be rounded using the C library's current rounding direction. No range checking of the input *data* is performed. It is up to you to ensure your data is within the limits of the output range you are using.

Return value

Comedi sample value corresponding to input physical value.

5.4.1.14 comedi_get_board_name

comedi_get_board_name — Comedi device name

Synopsis

```
#include <comedilib.h>
```

const 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 comedi device represented by *device*. This pointer is valid until the device is closed. This function returns NULL if there is an error.

5.4.1.15 comedi_get_driver_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.

5.4.1.16 comedi_get_maxdata

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 *channel* of subdevice *subdevice* belonging to the comedi device *device*.

Return value

The maximum valid sample value, or 0 on error.

5.4.1.17 comedi_get_n_channels

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 *`subdevice`* belonging to the comedi device *`device`*. This function returns `-1` on error and the Comedilib error value is set.

5.4.1.18 comedi_get_n_ranges

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 *`channel`* belonging to the subdevice *`subdevice`* of the comedi device *`device`*. This function returns `-1` on error.

5.4.1.19 comedi_get_n_subdevices

comedi_get_n_subdevices — number of subdevices

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_n_subdevices**(comedi_t * device);

Description

The function `comedi_get_n_subdevices()` returns the number of subdevices belonging to the Comedi device referenced by the parameter *`device`*, or `-1` on error.

5.4.1.20 comedi_get_range

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 on the range specified by the *`subdevice`*, *`channel`*, and *`range`* parameters. The pointer is valid until the Comedi device *`device`* is closed. If there is an error, `NULL` is returned.

5.4.1.21 comedi_get_subdevice_flags

comedi_get_subdevice_flags — properties of subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_subdevice_flags**(comedi_t * device, unsigned int subdevice);

Description

The function `comedi_get_subdevice_flags()` returns a bitfield describing the capabilities of the specified subdevice *`subdevice`* of the Comedi device *`device`*. If there is an error, `-1` is returned, and the Comedilib error value is set.

Subdevice Flag	Value (hex)	Description
SDF_BUSY	0x00000001	The subdevice is busy performing an asynchronous command. A subdevice being "busy" is slightly different from the "running" state flagged by SDF_RUNNING. A "running" subdevice is always "busy", but a "busy" subdevice is not necessarily "running". For example, suppose an analog input command has been completed by the hardware, but there are still samples in Comedi's buffer waiting to be read out. In this case, the subdevice is not "running", but is still "busy" until all the samples are read out or comedi_cancel() is called.
SDF_BUSY_OWNER	0x00000002	The subdevice is "busy", and the command it is running was started by the current process.
SDF_LOCKED	0x00000004	The subdevice has been locked by comedi_lock().
SDF_LOCK_OWNER	0x00000008	The subdevice is locked, and was locked by the current process.
SDF_MAXDATA	0x00000010	The maximum data value for the subdevice depends on the channel.
SDF_FLAGS	0x00000020	The subdevice flags depend on the channel (unfinished/broken support in library).
SDF_RANGETYPE	0x00000040	The range type depends on the channel.
SDF_CMD	0x00001000	The subdevice supports asynchronous commands.
SDF_SOFT_CALIBRATED	0x00002000	The subdevice relies on the host to do calibration in software. Software calibration coefficients are determined by the **comedi_soft_calibrate** utility. See the description of the comedi_get_softcal_converter() function for more information.
SDF_READABLE	0x00010000	The subdevice can be read (e.g. analog input).

Subdevice Flag	Value (hex)	Description
SDF_WRITABLE	0x00020000	The subdevice can be written to (e.g. analog output).
SDF_INTERNAL	0x00040000	The subdevice does not have externally visible lines.
SDF_GROUND	0x00100000	The subdevice supports analog reference AREF_GROUND.
SDF_COMMON	0x00200000	The subdevice supports analog reference AREF_COMMON.
SDF_DIFF	0x00400000	The subdevice supports analog reference AREF_DIFF.
SDF_OTHER	0x00800000	The subdevice supports analog reference AREF_OTHER
SDF_DITHER	0x01000000	The subdevice supports dithering (via the CR_ALT_FILTER chanspec flag).
SDF_DEGLITCH	0x02000000	The subdevice supports deglitching (via the CR_ALT_FILTER chanspec flag).
SDF_RUNNING	0x08000000	An asynchronous command is running. You can use this flag to poll for the completion of an output command.
SDF_LSAMPL	0x10000000	The subdevice uses the 32-bit lsampl_t type instead of the 16-bit sampl_t for asynchronous command data.
SDF_PACKED	0x20000000	The subdevice uses bitfield samples for asynchronous command data, one bit per channel (otherwise it uses one sampl_t or lsampl_t per channel). Commonly used for digital subdevices.

5.4.1.22 comedi_get_subdevice_type

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 subdevice index *subdevice*. The comedi_subdevice_type enum specifies the possible values for the subdevice type.

Return value

The function returns the subdevice type, or -1 if there is an error.

5.4.1.23 comedi_get_version_code

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.

5.4.1.24 comedi_internal_trigger

comedi_internal_trigger — generate soft trigger

Synopsis

```
#include <comedilib.h>
```

int **comedi_internal_trigger**(comedi_t * device, unsigned int subdevice, unsigned int trig_num);

Description

This function sends an INSN_INTTRIG instruction to a subdevice, which causes an internal triggering event. This event can, for example, trigger a subdevice to start an asynchronous command.

The *trig_num* parameter is reserved for future use, and should be set to 0. It is likely it will be used in the future to support multiple independent internal triggers. For example, an asynchronous command might be specified for a subdevice with a *start_src* of TRIG_INT, and a *start_arg* of 5. Then the start event would only be triggered if comedi_internal_trigger() were called on the subdevice with a *trig_num* equal to the same value of 5.

Return value

0 on success, -1 on error.

5.4.1.25 comedi_lock

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 by comedi_unlock(), or when comedi_close() is called on *device*.

Return value

If successful, `0` is returned. If there is an error, `-1` is returned.

5.4.1.26 comedi_maxdata_is_chan_specific

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 may have different maximum sample values, this function returns `1`. Otherwise, this function returns `0`. On error, this function returns `-1`.

5.4.1.27 comedi_open

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 successful, `comedi_open()` returns a pointer to a valid comedi_t structure. This structure is opaque; the pointer should not be dereferenced by the application. `NULL` is returned on failure.

5.4.1.28 comedi_range_is_chan_specific

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 may have different range information, this function returns `1`. Otherwise, this function returns `0`. On error, this function returns `-1`.

5.4.1.29 comedi_set_global_oor_behavior

comedi_set_global_oor_behavior — out-of-range behavior

Synopsis

```
#include <comedilib.h>
```

enum comedi_oor_behavior **comedi_set_global_oor_behavior**(enum comedi_oor_behavior behavior);

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.

Return value

The previous out-of-range behavior is returned.

5.4.1.30 comedi_to_phys

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 parameter *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. The mapping between physical units is linear and assumes ideal converter characteristics.

Conversion of endpoint sample values, that is, sample values equal to 0 or *maxdata*, is affected by the Comedilib out-of-range behavior (see function comedi_set_global_oor_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.

5.4.1.31 comedi_to_physical

comedi_to_physical — convert sample to physical units using polynomials

Synopsis

```
#include <comedilib.h>
```

double **comedi_to_physical**(lsampl_t data, const comedi_polynomial_t * conversion_polynomial);

Description

Converts *data* given in Comedi's integer sample values (lsampl_t, between 0 and maxdata) into physical units (double). The *conversion_polynomial* parameter is obtained from either comedi_get_hardcal_converter() or comedi_get_softcal_converter(). No range checking of the input *data* is performed. It is up to you to check for *data* values of 0 or maxdata if you want to detect possibly out-of-range readings.

Return value

Physical value corresponding to the input sample value.

5.4.1.32 comedi_unlock

comedi_unlock — subdevice reservation

Synopsis

```
#include <comedilib.h>
```

int **comedi_unlock**(comedi_t * device, unsigned int subdevice);

Description

The function comedi_unlock() releases a subdevice locked by comedi_lock().

Return value

0 on success, otherwise -1.

5.4.2 Asynchronous commands

5.4.2.1 comedi_cancel

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 command previously started by comedi_command() which is still in progress on the subdevice indicated by the parameters *device* and *subdevice*.

Return value

On success, 0 is returned. On failure, -1 is returned.

5.4.2.2 comedi_command

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 a streaming input or output. The command structure pointed to by *command* specifies settings for 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() on the device file descriptor. For output subdevices, sample values are written using the function write().

Return value

On success, 0 is returned. On failure, -1 is returned.

5.4.2.3 comedi_command_test

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 successfully passed. In addition, if elements of the comedi_cmd 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.

Return 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 analog input boards require that all channels in the chanlist use the same input range.

On failure, -1 is returned.

5.4.2.4 comedi_get_buffer_contents

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 to get the number of unread bytes. For a streaming input command, this is the number of bytes that can be read. For a streaming output command, subtracting this from the buffer size gives the space available to be written.

Return value

On success, `comedi_get_buffer_contents()` returns the number of unread bytes in the buffer. On failure, `-1` is returned.

5.4.2.5 comedi_get_buffer_read_offset

comedi_get_buffer_read_offset — streaming buffer read offset

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_buffer_read_offset**(comedi_t * device, unsigned int subdevice);

Description

The function `comedi_get_buffer_read_offset()` is used on a subdevice that has a Comedi command in progress to get the current read position in the streaming buffer as an offset in bytes from the start of the buffer. The position will wrap around to 0 when it reaches the buffer size. This offset is only useful for memory mapped buffers.

This function replaces `comedi_get_buffer_offset()` and has the same functionality.

Return value

On success, `comedi_get_buffer_read_offset()` returns the current read position as an offset in bytes from the start of the buffer. On failure, `-1` is returned.

5.4.2.6 comedi_get_buffer_write_offset

comedi_get_buffer_write_offset — streaming buffer write offset

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_buffer_write_offset**(comedi_t * device, unsigned int subdevice);

Description

The function `comedi_get_buffer_write_offset()` is used on a subdevice that has a Comedi command in progress to get the current write position in the streaming buffer as an offset in bytes from the start of the buffer. The position will wrap around to 0 when it reaches the buffer size. This offset is only useful for memory mapped buffers.

Return value

On success, `comedi_get_buffer_write_offset()` returns the current write position as an offset in bytes from the start of the buffer. On failure, `-1` is returned.

5.4.2.7 comedi_get_buffer_read_count

comedi_get_buffer_read_count — streaming buffer read count

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_buffer_read_count**(comedi_t * device, unsigned int subdevice, unsigned int * read_count);

Description

The function `comedi_get_buffer_read_count()` is used on a subdevice that has a Comedi command in progress to get the number of bytes that have been read from the buffer, modulo `UINT_MAX + 1`. The value is stored in *`*read_count`*.

Return value

On success, `0` is returned. On failure, `-1` is returned.

5.4.2.8 comedi_get_buffer_write_count

comedi_get_buffer_write_count — streaming buffer write count

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_buffer_write_count**(comedi_t * device, unsigned int subdevice, unsigned int * write_count);

Description

The function `comedi_get_buffer_write_count()` is used on a subdevice that has a Comedi command in progress to get the number of bytes that have been written to the buffer, modulo `UINT_MAX + 1`. The value is stored in *`*write_count`*.

Return value

On success, `0` is returned. On failure, `-1` is returned.

5.4.2.9 comedi_get_buffer_size

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*.

Return value

On success, comedi_get_buffer_size() returns the size of the buffer in bytes. On failure, -1 is returned.

5.4.2.10 comedi_get_cmd_generic_timed

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 chanlist_len, unsigned int scan_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 *command* is modified to be a valid command that can be used as a parameter to comedi_command() (after the command has additionally been assigned a valid *chanlist* array). The command measures scans consisting of *chanlist_len* channels at a scan rate that corresponds to a period of *scan_period_ns* nanoseconds. The rate is adjusted to a rate that the device can handle.

Return value

On success, 0 is returned. On failure, -1 is returned.

5.4.2.11 comedi_get_cmd_src_mask

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 *command*. The trigger source elements of the command structure are set to be the bitwise-or of the subdevice's supported trigger sources. Other elements in the structure are undefined.

Return value

On success, `0` is returned. On failure, `-1` is returned.

5.4.2.12 comedi_get_max_buffer_size

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 can be accomplished with `comedi_set_max_buffer_size`() or with the **comedi_config** program, and requires appropriate privileges.

Return value

On success, the maximum allowable size (in bytes) of the streaming buffer is returned. On failure, `-1` is returned.

5.4.2.13 comedi_get_read_subdevice

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 index of the subdevice whose streaming input buffer is currently accessible through the device *`device`*, if there is one.

Return value

On success, `comedi_get_read_subdevice`()returns the index of the current “read” subdevice if there is one, or `-1` if there is no “read” subdevice. On failure, `-1` is returned.

5.4.2.14 comedi_get_write_subdevice

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 index of the subdevice whose streaming output buffer is currently accessible through the device *device*, if there is one.

Return value

On success, comedi_get_write_subdevice()returns the index of the current "write" subdevice if there is one, or -1 if there is no "write" subdevice. On failure, -1 is returned.

5.4.2.15 comedi_mark_buffer_read

comedi_mark_buffer_read — streaming buffer control

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 input command in progress. It should only be used if you are using a mmap() mapping to read data from Comedi's buffer (as opposed to calling read() on the device file descriptor), since Comedi will automatically keep track of how many bytes have been transferred via read() calls. This function is used to indicate that the next *num_bytes* bytes in the buffer are no longer needed and may be discarded.

Return value

On success, comedi_mark_buffer_read() returns the number of bytes successfully marked as read. The return value may be less than the *num_bytes*. On failure, -1 is returned.

5.4.2.16 comedi_mark_buffer_written

comedi_mark_buffer_written — streaming buffer control

Synopsis

```
#include <comedilib.h>
```

int **comedi_mark_buffer_written**(comedi_t * device, unsigned int subdevice, unsigned int num_bytes);

Description

The function comedi_mark_buffer_written() is used on a subdevice that has a Comedi output command in progress. It should only be used if you are using a mmap() mapping to write data to Comedi's buffer (as opposed to calling write() on the device file descriptor), since Comedi will automatically keep track of how many bytes have been transferred via write() calls. This function is used to indicate that the next *num_bytes* bytes in the buffer are valid and may be sent to the device.

Return value

On success, comedi_mark_buffer_written() returns the number of bytes successfully marked as written. The return value may be less than the *num_bytes*. On failure, -1 is returned.

5.4.2.17 comedi_poll

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. Only a few Comedi drivers support this operation.

Return value

On success, `comedi_poll()` returns the number of additional bytes available. On failure, `-1` is returned.

5.4.2.18 comedi_set_buffer_size

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 buffer size will be set to *`size`* bytes, rounded up to a multiple of the virtual memory page size. The virtual memory page size can be determined using `sysconf(_SC_PAGE_SIZE)`.

This function does not require special privileges. However, it is limited to a (adjustable) maximum buffer size, which can be changed by a privileged user calling `comedi_set_max_buffer_size()`, or running the program **comedi_config**.

Return value

On success, `comedi_set_buffer_size()` returns the new buffer size in bytes. On failure, `-1` is returned.

5.4.2.19 comedi_set_max_buffer_size

comedi_set_max_buffer_size — streaming maximum 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 the user to have appropriate privileges.

Return value

On success, `comedi_set_max_buffer_size()` returns the new maximum buffer size in bytes. On failure, `-1` is returned.

5.4.2.20 comedi_set_read_subdevice

comedi_set_read_subdevice — set streaming input subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_set_read_subdevice**(comedi_t * device, unsigned int subdevice);

Status

Works for Linux "in-tree" Comedi since kernel version 3.19.

Description

The function `comedi_set_read_subdevice()` sets *`subdevice`* as the current “read” subdevice if the subdevice supports streaming input commands.

No action is performed if *`subdevice`* is already the current “read” subdevice.

Changes are local to the *open file description* for this *`device`* and have no effect on other open file descriptions for the underlying device node.

Return value

On success, `0` is returned. On failure, `-1` is returned.

5.4.2.21 comedi_set_write_subdevice

comedi_set_write_subdevice — set streaming output subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_set_write_subdevice**(comedi_t * device, unsigned int subdevice);

Status

Works for Linux "in-tree" Comedi since kernel version 3.19.

Description

The function comedi_set_write_subdevice() sets *subdevice* as the current "write" subdevice if the subdevice supports streaming output commands.

No action is performed if *subdevice* is already the current "write" subdevice.

Changes are local to the *open file description* for this *device* and have no effect on other open file descriptions for the underlying device node.

Return value

On success, 0 is returned. On failure, −1 is returned.

5.4.3 Calibration

5.4.3.1 comedi_apply_calibration

comedi_apply_calibration — set hardware 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

The function comedi_apply_calibration() sets the hardware calibration for the subdevice specified by *device* and *subdevice* so that it is in proper calibration when using the channel specified by *channel*, range index specified by *range* and analog reference specified by *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 analog reference. Furthermore, the calibrations appropriate for different channel, range, and analog reference parameters may not be able to be applied simultaneously. For example, some boards cannot have their analog inputs calibrated for more than one input range simultaneously. Applying a calibration for range 1 may blow away a previously applied calibration for range 0. Or, 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 analog reference. 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 calibration 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

Returns `0` on success, `-1` on failure.

5.4.3.2 comedi_apply_parsed_calibration

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 value of parameter *`calibration`* is obtained by a call to `comedi_parse_calibration_file`().

Return value

Returns `0` on success, `-1` on failure.

5.4.3.3 comedi_cleanup_calibration

comedi_cleanup_calibration — free calibration resources

Synopsis

```
#include <comedilib.h>
```

void **comedi_cleanup_calibration**(comedi_calibration_t * calibration);

Status

alpha

Description

This function frees the resources associated with a comedi_calibration_t obtained from `comedi_parse_calibration_file`(). The comedi_calibration_t pointed to by *`calibration`* can not be used again after calling this function.

5.4.3.4 comedi_get_default_calibration_path

comedi_get_default_calibration_path — get default calibration file path

Synopsis

```
#include <comedilib.h>
```

char * **comedi_get_default_calibration_path**(comedi_t * device);

Status

alpha

Description

This function returns a pointer to a string containing a default calibration file path appropriate for the Comedi device specified by *device*. Memory for the string is allocated by the function, and should be freed with the C library function free() 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.

5.4.3.5 comedi_get_hardcal_converter

comedi_get_hardcal_converter — get converter for hardware-calibrated subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_hardcal_converter**(comedi_t * device, unsigned subdevice, unsigned channel, unsigned range, enum comedi_conversio
direction, comedi_polynomial_t * converter);

Status

alpha

Description

The function comedi_get_hardcal_converter() initializes the comedi_polynomial_t pointed to by *converter* so it can be passed to either comedi_to_physical(), or comedi_from_physical(). The result can be used to convert data from the specified *subdevice*, *channel*, and *range*. The *direction* parameter specifies whether *converter* will be passed to comedi_to_physical() or comedi_from_physical().

This function initializes the comedi_polynomial_t pointed to by *converter* as a simple linear function with no calibration information, appropriate for boards which do their gain/offset/nonlinearity corrections in hardware. If your board needs calibration to be performed in software by the host computer, use comedi_get_softcal_converter() instead. A subdevice will advertise the fact that it depends on a software calibration with the SDF_SOFT_CALIBRATED subdevice flag.

The result of this function will only depend on the *channel* parameter if either comedi_range_is_chan_specific() or comedi_maxdata_is_chan_specific() returns true for the specified *subdevice*.

Return value

Returns 0 on success, -1 on failure.

5.4.3.6 comedi_get_softcal_converter

comedi_get_softcal_converter — get converter for software-calibrated subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_softcal_converter**(unsigned subdevice, unsigned channel, unsigned range, enum comedi_conversion_direction direction, const comedi_calibration_t * parsed_calibration, comedi_polynomial_t * converter);

Status

alpha

Description

The function `comedi_get_softcal_converter()` initializes the comedi_polynomial_t pointed to by *`converter`* so it can be passed to either `comedi_to_physical()` or `comedi_from_physical()`. The comedi_polynomial_t pointed to by *`converter`* can then be used to convert data for the specified *`subdevice`*, *`channel`*, and *`range`*. The *`direction`* parameter specifies whether *`converter`* will be passed to `comedi_to_physical()` or `comedi_from_physical()`. The *`parsed_calibration`* parameter points to the software calibration values for your device, and may be obtained by calling `comedi_parse_calibration_file()` on a calibration file generated by the **comedi_soft_calibrate** program.

This function is only useful for boards that perform their calibrations in software on the host computer. A subdevice will advertise the fact that it depends on a software calibration with the `SDF_SOFT_CALIBRATED` subdevice flag.

Whether or not the result of this function actually depends on the *`channel`* parameter is hardware dependent. For example, the calibration of a multiplexed analog input will typically not depend on the channel, only the range. Analog outputs will typically use different calibrations for each output channel.

Software calibrations are implemented as polynomials (up to third order). Since the inverse of a polynomial of order higher than one can't be represented exactly as another polynomial, you may not be able to get converters for the "reverse" direction. For example, you may be able to get a converter for an analog input in the `COMEDI_TO_PHYSICAL` direction, but not in the `COMEDI_FROM_PHYSICAL` direction.

Return value

Returns `0` on success, `-1` on failure.

5.4.3.7 comedi_parse_calibration_file

comedi_parse_calibration_file — load contents of calibration file

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** or **comedi_soft_calibrate** programs) and returns a pointer to a comedi_calibration_t which can be passed to the comedi_apply_parsed_calibration() or comedi_get_softcal_converter() functions. 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.

5.4.4 Digital I/O

5.4.4.1 comedi_dio_bitfield2

comedi_dio_bitfield2 — read/write multiple digital channels

Synopsis

```
#include <comedilib.h>
```

int **comedi_dio_bitfield2**(comedi_t * device, unsigned int subdevice, unsigned int write_mask, unsigned int * bits, unsigned int base_channel);

Description

The function comedi_dio_bitfield2() allows multiple channels to be read or written together on a digital input, output, or configurable 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 *base_channel*. For each bit in *write_mask* that is set to 1, the corresponding 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 the last value written to the output channel.

All the channels might not be read or written at the exact same time. For example, the driver may need to sequentially write to several registers in order to set all the digital channels specified by the *write_mask* and *base_channel* parameters.

Return value

If successful, 0 is returned, otherwise -1.

5.4.4.2 comedi_dio_config

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 *direction*. Valid directions are COMEDI_INPUT or COMEDI_OUTPUT.

Depending on the characteristics of the hardware device, multiple channels might be grouped together in hardware when configuring the input/output direction. In this case, a single call to comedi_dio_config() for any channel in the group will affect the entire group.

Return value

If successful, 0 is returned, otherwise -1.

5.4.4.3 comedi_dio_get_config

comedi_dio_get_config — query input/output properties of channel

Synopsis

```
#include <comedilib.h>
```

int **comedi_dio_get_config**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int * direction);

Description

The function comedi_dio_get_config() queries the input/output configuration of an individual channel in a digital I/O subdevice (see comedi_dio_config()). On success, **direction* will be set to either COMEDI_INPUT or COMEDI_OUTPUT.

Return value

If successful, 0 is returned, otherwise -1.

5.4.4.4 comedi_dio_read

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 comedi_dio_read() reads the channel *channel* belonging to the subdevice *subdevice* of device *device*. The data value that is read is stored in the **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.

If you wish to read multiple digital channels at once, it is more efficient to use comedi_dio_bitfield2() than to call this function multiple times.

Return value

Return values and errors are the same as comedi_data_read().

5.4.4.5 comedi_dio_write

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.

If you wish to write multiple digital channels at once, it is more efficient to use comedi_dio_bitfield2() than to call this function multiple times.

Return value

Return values and errors are the same as comedi_data_write().

5.4.5 Error reporting

5.4.5.1 comedi_errno

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 by calling 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().

5.4.5.2 comedi_loglevel

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 log level *loglevel*, additional debugging information will be printed. Error and debugging messages are printed to the standard error output stream stderr.

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

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

The meaning of the log levels is as follows:

Loglevel	Behavior
0	Comedilib prints nothing.
1	(default) Comedilib prints error messages when there is a self-consistency error (i.e., an internal bug.)
2	Comedilib prints an error message when an invalid parameter is passed.
3	Comedilib prints an error message whenever an error is generated in the Comedilib library or in the C library, when called by Comedilib.
4	Comedilib prints a lot of junk.

Return value

This function returns the previous log level.

5.4.5.3 comedi_perror

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() or 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 the standard error output stream stderr. The error message consists of the argument string *s*, a colon, a space, a description of the error condition, and a new line.

5.4.5.4 comedi_strerror

comedi_strerror — return string describing Comedilib error code

Synopsis

```
#include <comedilib.h>
```

const 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`() or `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 returned string may be modified by a subsequent call to a `strerr`() or `perror`() function (either the `libc` or Comedilib versions). An unrecognized error number will return a pointer to the string "undefined error", or similar.

5.4.6 Extensions

5.4.6.1 comedi_arm

comedi_arm — arm a subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_arm**(comedi_t * device, unsigned int subdevice, unsigned int source);

Status

alpha

Description

This function arms a subdevice. It may, for example, arm a counter to begin counting. The *`source`* parameter specifies what source should trigger the subdevice to begin. The possible sources are driver-dependant. This function is only useable on subdevices that provide support for the `INSN_CONFIG_ARM` configuration instruction. Some subdevices treat this as an instruction to arm a specific channel. For those subdevices, this function will arm channel `0` and `comedi_arm_channel`() should be called instead of this one to specify the channel.

Return value

`0` on success, `-1` on error.

5.4.6.2 comedi_arm_channel

comedi_arm_channel — arm a subdevice channel

Synopsis

```
#include <comedilib.h>
```

int **comedi_arm_channel**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int source);

Status

alpha

Description

This function arms a specified channel of a subdevice. It may, for example, arm a counter to begin counting. The *source* parameter specifies what source should trigger the subdevice to begin. The possible sources are driver-dependant. This function is only useable on subdevices that provide support for the INSN_CONFIG_ARM configuration instruction. Some subdevices treat this as an instruction to arm the whole subdevice and ignore the specified channel. For such subdevices, comedi_arm() is normally called instead.

Return value

0 on success, -1 on error.

5.4.6.3 comedi_digital_trigger_disable

comedi_digital_trigger_disable — disable a digital trigger

Synopsis

```
#include <comedilib.h>
```

int **comedi_digital_trigger_disable**(comedi_t * device, unsigned int subdevice, unsigned int trigger_id);

Status

alpha

Description

This function disables a digital trigger on a subdevice, returning it to its default, inactive, unconfigured state. If the subdevice supports several digital triggers, the *trigger_id* selects one.

This function is only useable on subdevices that provide support for the INSN_CONFIG_DIGITAL_TRIG configuration instruction.

Return value

0 on success, -1 on error.

5.4.6.4 comedi_digital_trigger_enable_edges

comedi_digital_trigger_enable_edges — set digital trigger edge detection

Synopsis

```
#include <comedilib.h>
```

int **comedi_digital_trigger_enable_edges**(comedi_t * device, unsigned int subdevice, unsigned int trigger_id, unsigned int base_input, unsigned int rising_edge_inputs, unsigned int falling_edge_inputs);

Status

alpha

Description

This function enables edge detection for a digital trigger on a subdevice. If the subdevice supports several digital triggers, the *trigger_id* selects one. The *rising_edge_inputs* and *falling_edge_inputs* parameters are bit fields that enable (1) or disable (0) rising and falling edge detection on a set of (up to) 32 inputs. The least-significant bit corresponds to the input specified by the *base_input* parameter, with subsequent bits corresponding to subsequent inputs.

Successive calls to this function have an cumulative effect, which allows digital triggers to be set up for more than 32 inputs. There may also be a cumulative effect with calls to comedi_digital_trigger_enable_levels() if the digital trigger supports a combination of edge and level triggering. Due to the cumulative effect, it may be necessary to call comedi_digital_trigger_disable() to clear the old settings before reconfiguring the digital trigger inputs.

A digital trigger may support edge detection, level detection, both at different times, or both at the same time. If it supports both but not at the same time, configuring edge triggers will disable any previous level triggers, or vice versa.

This function is only useable on subdevices that provide support for the INSN_CONFIG_DIGITAL_TRIG configuration instruction, and only if the digital trigger supports edge detection.

Return value

0 on success, -1 on error.

5.4.6.5 comedi_digital_trigger_enable_levels

comedi_digital_trigger_enable_levels — set digital trigger level detection

Synopsis

```
#include <comedilib.h>
```

int **comedi_digital_trigger_enable_levels**(comedi_t * device, unsigned int subdevice, unsigned int trigger_id, unsigned int base_input, unsigned int high_level_inputs, unsigned int low_level_inputs);

Status

alpha

Description

This function enables level detection for a digital trigger on a subdevice. If the subdevice supports several digital triggers, the *trigger_id* selects one. The *high_level_inputs* and *low_level_inputs* parameters are bit fields that enable (1) or disable (0) high and low level detection on a set of (up to) 32 inputs. The least-significant bit corresponds to the input specified by the *base_input* parameter, with subsequent bits corresponding to subsequent inputs.

Successive calls to this function have an cumulative effect, which allows digital triggers to be set up for more than 32 inputs. There may also be a cumulative effect with calls to comedi_digital_trigger_enable_edges() if the digital trigger supports a combination of edge and level triggering. Due to the cumulative effect, it may be necessary to call comedi_digital_trigger_disable() to clear the old settings before reconfiguring the digital trigger inputs.

A digital trigger may support edge detection, level detection, both at different times, or both at the same time. If it supports both but not at the same time, configuring level triggers will disable any previous edge triggers, or vice versa.

This function is only useable on subdevices that provide support for the INSN_CONFIG_DIGITAL_TRIG configuration instruction, and only if the digital trigger supports level detection.

Return value

0 on success, -1 on error.

5.4.6.6 comedi_disarm

comedi_disarm — disarm a subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_disarm**(comedi_t * device, unsigned int subdevice);

Status

alpha

Description

This function disarms a subdevice. It may, for example, stop a counter counting. This function is only useable on subdevices that provide support for the INSN_CONFIG_DISARM configuration instruction. Some subdevices treat this as an instruction to disarm a specific channel. For those subdevices, this function will disarm channel 0 and comedi_disarm_channel() should be called instead of this one to specify the channel.

Return value

0 on success, -1 on error.

5.4.6.7 comedi_disarm_channel

comedi_disarm_channel — disarm a subdevice channel

Synopsis

```
#include <comedilib.h>
```

int **comedi_disarm_channel**(comedi_t * device, unsigned int subdevice, unsigned int channel);

Status

alpha

Description

This function disarms a specified channel of a subdevice. It may, for example, stop a counter counting. This function is only useable on subdevices that provide support for the `INSN_CONFIG_DISARM` configuration instruction. Some subdevices treat this as an instruction to disarm the whole subdevice and ignore the specified channel. For such subdevices, `comedi_disarm`() is normally called instead.

Return value

`0` on success, `-1` on error.

5.4.6.8 comedi_get_clock_source

comedi_get_clock_source — get master clock for a subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_clock_source**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int * clock, unsigned int * period_ns);

Status

alpha

Description

This function queries the master clock for a subdevice, as set by `comedi_set_clock_source`(). The currently configured master clock will be written to *`*clock`*. The possible values and their corresponding clocks are driver-dependant. The period of the clock in nanoseconds (or zero if it is unknown) will be written to *`*period_ns`*. If the subdevice does not support configuring its master clocks on a per-channel basis, then the *`channel`* parameter will be ignored.

It is safe to pass `NULL` pointers as the *`clock`* or *`period_ns`* parameters. This function is only useable on subdevices that provide support for the `INSN_CONFIG_GET_CLOCK_SRC` configuration instruction.

Return value

`0` on success, `-1` on error.

5.4.6.9 comedi_get_gate_source

comedi_get_gate_source — get gate for a subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_gate_source**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int gate_index, unsigned int * gate_source);

Status

alpha

Description

This function queries the gate for a subdevice, as set by comedi_set_gate_source(). The currently configured gate source will be written to *gate_source. The possible values and their corresponding gates are driver-dependant. If the subdevice does not support configuring its gates on a per-channel basis, then the channel parameter will be ignored.

This function is only useable on subdevices that provide support for the INSN_CONFIG_GET_GATE_SRC configuration instruction.

Return value

0 on success, -1 on error.

5.4.6.10 comedi_get_hardware_buffer_size

comedi_get_hardware_buffer_size — get size of subdevice's hardware buffer

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_hardware_buffer_size**(comedi_t *device, unsigned int subdevice, enum comedi_io_direction direction);

Description

This function returns the number of bytes the subdevice can hold in it's hardware buffer. The term "hardware buffer" refers to any FIFOs, etc. on the acquisition board itself which are used during streaming commands. This does not include the buffer maintained by the comedi kernel module in host memory, whose size may be queried by comedi_get_buffer_size(). The direction parameter of type enum comedi_io_direction should be set to COMEDI_INPUT to query the input buffer size (e.g., the buffer of an analog input subdevice), or COMEDI_OUTPUT to query the output buffer size (e.g., the buffer of an analog output).

Return value

Hardware buffer size in bytes, or -1 on error.

5.4.6.11 comedi_get_routing

comedi_get_routing — get routing for an output

Synopsis

```
#include <comedilib.h>
```

int **comedi_get_routing**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int * routing);

Status

alpha

Description

This function queries the routing for an output, as set by comedi_set_routing(). The currently configured routing will be written to *routing. The possible values and their corresponding routings are driver-dependant.

This function is only useable on subdevices that provide support for the INSN_CONFIG_GET_ROUTING configuration instruction.

Return value

0 on success, -1 on error.

5.4.6.12 comedi_reset

comedi_reset — reset a subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_reset**(comedi_t * device, unsigned int subdevice);

Status

alpha

Description

This function resets a subdevice. It is only useable on subdevices that provide support for the INSN_CONFIG_RESET configuration instruction. Some subdevices treat this as an instruction to reset a specific channel. For those subdevices, this function will reset channel 0 and comedi_reset_channel() should be called instead of this one to specify the channel.

Return value

0 on success, -1 on error.

5.4.6.13 comedi_reset_channel

comedi_reset_channel — reset a subdevice channel

Synopsis

```
#include <comedilib.h>
```

int **comedi_reset_channel**(comedi_t * device, unsigned int subdevice, unsigned int channel);

Status

alpha

Description

This function resets a specified channel of a subdevice. It is only useable on subdevices that provide support for the INSN_CONFIG_RESET configuration instruction. Some subdevices treat this as an instruction to reset the whole subdevice and ignore the specified channel. For such subdevices, comedi_reset() is normally called instead.

Return value

0 on success, -1 on error.

5.4.6.14 comedi_set_clock_source

comedi_set_clock_source — set master clock for a subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_set_clock_source**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int clock, unsigned int period_ns);

Status

alpha

Description

This function selects a master clock for a subdevice. The *clock* parameter selects the master clock, and is driver-dependant. If the subdevice does not support configuring its master clocks on a per-channel basis, then the *channel* parameter will be ignored. The *period_ns* parameter specifies the clock's period in nanoseconds. It may left unspecified by using a value of zero. Drivers will ignore the clock period if they already know what the clock period should be for the specified clock (e.g. for an on-board 20MHz oscillator). Certain boards which use a phase-locked loop to synchronize to external clock sources must be told the period of the external clock. Specifying a clock period for an external clock may also allow the driver to support TRIG_TIMER sources in commands while using the external clock.

The clock may be queried with the comedi_get_clock_source() function.

This function is only useable on subdevices that provide support for the INSN_CONFIG_SET_CLOCK_SRC configuration instruction.

Return value

0 on success, -1 on error.

5.4.6.15 comedi_set_counter_mode

comedi_set_counter_mode — change mode of a counter subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_set_counter_mode**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int mode);

Status

alpha

Description

This function configures a counter subdevice. The meaning of the *mode* parameter is driver-dependent. If the subdevice does not support configuring its mode on a per-channel basis, then the *channel* parameter will be ignored.

It is only useable on subdevices that provide support for the INSN_CONFIG_SET_COUNTER_MODE configuration instruction.

Return value

0 on success, -1 on error.

5.4.6.16 comedi_set_filter

comedi_set_filter — select a filter for a subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_set_filter**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int filter);

Status

alpha

Description

This function selects a filter for a subdevice. For instance, a digital input subdevice may provide deglitching filters with varying cutoff frequencies. The filters are used to prevent high-frequency noise from causing unwanted transitions on the digital inputs. This function can tell the hardware which deglitching filter to use, or to use none at all.

The *filter* parameter selects which of the subdevice's filters to use, and is driver-dependant.

This function is only useable on subdevices that provide support for the INSN_CONFIG_FILTER configuration instruction.

Return value

0 on success, -1 on error.

5.4.6.17 comedi_set_gate_source

comedi_set_gate_source — select gate source for a subdevice

Synopsis

```
#include <comedilib.h>
```

int **comedi_set_gate_source**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int gate_index, unsigned int gate_source);

Status

alpha

Description

This function selects a gate source for a subdevice. The *gate_index* parameter selects which gate is being configured, should the subdevice have multiple gates. It takes a value from 0 to N-1 for a subdevice with N different gates. The *gate_source* parameter selects which signal you wish to use as the gate, and is also driver-dependent. If the subdevice does not support configuring its gates on a per-channel basis, then the *channel* parameter will be ignored.

You may query the gate source with the comedi_get_gate_source() function. This function is only useable on subdevices that provide support for the INSN_CONFIG_SET_GATE_SRC configuration instruction.

Return value

0 on success, -1 on error.

5.4.6.18 comedi_set_other_source

comedi_set_other_source — select source signal for something other than a gate or clock

Synopsis

```
#include <comedilib.h>
```

int **comedi_set_other_source**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int other, unsigned int source);

Status

alpha

Description

This function allows selection of a source signal for something on a subdevice other than a gate (which uses comedi_set_gate_source()) or a clock (which uses comedi_set_clock_source()). The *other* parameter selects which "other" we are configuring, and is driver-dependent. The *source* parameter selects the source we which to use for the "other". If the subdevice does not support configuring its "other" sources on a per-channel basis, then the *channel* parameter will be ignored.

As an example, this function is used to select which PFI digital input channels should be used as the A/B/Z signals when running a counter on an NI M-Series board as a quadrature encoder. The *other* parameter selects either the A, B, or Z signal, and the *source* parameter is used to specify which PFI digital input channel the external A, B, or Z signal is physically connected to.

This function is only useable on subdevices that provide support for the INSN_CONFIG_SET_OTHER_SRC configuration instruction.

Return value

0 on success, -1 on error.

5.4.6.19 comedi_set_routing

comedi_set_routing — select a routing for an output

Synopsis

```
#include <comedilib.h>
```

int **comedi_set_routing**(comedi_t * device, unsigned int subdevice, unsigned int channel, unsigned int routing);

Status

alpha

Description

This function configures a mutiplexed output channel which can output a variety of different signals (such as NI's RTSI and PFI lines). The *routing* parameter selects which signal should be routed to appear on the selected output channel, and is driver-dependant.

The routing may be queried with the comedi_get_routing() function. This function is only useable on subdevices that provide support for the INSN_CONFIG_SET_ROUTING configuration instruction.

Return value

0 on success, -1 on error.

5.4.7 Deprecated functions

5.4.7.1 comedi_dio_bitfield

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);

Status

deprecated

Description

This function is deprecated. Use comedi_dio_bitfield2() instead. It is equivalent to using comedi_dio_bitfield2() with *base_channel* set to 0.

5.4.7.2 comedi_get_buffer_offset

comedi_get_buffer_offset — streaming buffer status (deprecated)

Status

deprecated

Description

This function is deprecated. Use comedi_get_buffer_read_offset() instead. It has the same functionality as comedi_get_buffer_read_offset().

5.4.7.3 comedi_get_timer

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 library. 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.

5.4.7.4 comedi_sv_init

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 pointed to by *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.

Return value

This function returns 0 on success, -1 on error.

5.4.7.5 comedi_sv_measure

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 pointed to by *`sv`* to measure a slowly varying signal. If successful, 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.

5.4.7.6 comedi_sv_update

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 pointed to by *`sv`*.

5.4.7.7 comedi_timed_1chan

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.

5.4.7.8 comedi_trigger

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 pointed to by *`trig`*. The return value depends on the particular trigger being issued. If there is an error, `-1` is returned.

5.5 Language bindings

Comedilib is a C library but comes also with bindings for Python, Perl, and Ruby. This enables usage of a DAQ device directly from these higher level languages.

5.5.1 Python bindings

Python bindings are automatically generated by SWIG. So always keep in mind that for precise information the SWIG documentation can come in handy in addition to the Comedi documentation.

The name of the module is called `comedi`. All the C functions, structs, and constants are directly available as is. So you can refer directly to the C documentation for most of the usage. Note that, as in C, the functions either return the successful result or an error code. Unlike typical Python functions, no exceptions are generated on errors (excepted type-checking performed by SWIG). For example, to open a Comedi device you can write in Python:

```
import comedi
device = comedi.comedi_open("/dev/comedi0")
if device is None:
    errno = comedi.comedi_errno()
    print "Error (%d) %s" % (errno, comedi.comedi_strerror(errno))
    return
```

There are a few things to be aware of. The SWIG bindings automatically take care of converting functions with output parameters to function returning multiple values, if the type of the output parameter is simple. For example `comedi_data_read()` takes as argument a *`lsampl_t *data`* which will contain the data read after the function returns. So in C it is used like this:

```
lsampl_t data;
rc = comedi_data_read(device, subdevice, channel, range, aref, &data);
```

As *`lsampl_t`* is a 32-bit *`unsigned int`*, in Python, the function is used like this:

```
rc, data = comedi.comedi_data_read(device, subdevice, channel, range, aref)
```

SWIG takes care of converting simple types between Python and C, but does not convert more complex types such as arrays or structs. Special Python classes are created for these. Moreover, Comedi also provides a few macros. These macros are available in Python as functions with similar names, but lowercase: `comedi.cr_pack()`, `comedi.cr_range()`, etc. So to create and modify a command, one can do in Python:

```
cmd = comedi.comedi_cmd_struct()
comedi.comedi_get_cmd_generic_timed(device, subdevice, cmd, nchans, period_ns)
cmd.stop_src = comedi.TRIG_COUNT
cmd.stop_arg = nscans
# create and add the channel list
clist = comedi.chanlist(nchans)
for i, channel in range(nchans):
    clist[i] = comedi.cr_pack(channel, range, comedi.AREF_GROUND)
cmd.chanlist = clist
```

One unfortunate consequence is that the objects to represent arrays are as low-level as C arrays (i.e., a pointer with an addition). They have no range checking and no iterator. In addition, they have to be `cast`() from the Python object to the C object. So to create an instruction to call the internal trigger, you would write in Python:

```
insn = comedi.comedi_insn_struct()
insn.subdev = subdevice
insn.insn = comedi.INSN_INTTRIG
insn.n = 1
data = comedi.lsampl_array(insn.n)
data[0] = num
insn.data = data.cast()
rc = comedi.comedi_do_insn(device, insn)
```

When you need to convert from a raw SWIG object to a proxy object, the `.frompointer`() of the Python object can be used. Also note that by default when the proxy object is deleted, SWIG frees the memory associated to it. To still be able to use the memory, you need to set *`.thisown`* to *`False`*.

Reading (or writing) a large set of data from a Comedi device is done via a file. In C, this is done by using a file number, but in Python you need a *`File`* object. You can get a *`File`* object via `os.fdopen`(). For example, to read an array of input data from a Comedi device into a numpy array, one could do:

```
fileno = comedi.comedi_fileno(device)
file = os.fdopen(fileno, 'r+')
buf = numpy.fromfile(file, dtype=numpy.uint32, count=(nscans * nchans))
```

5.6 Kernel drivers

5.6.1 8255 -- 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'.
```

5.6.2 acl7225b -- ADLINK NuDAQ ACL-7225b & compatibles

Author: José Luis Sánchez (jsanchezv@teleline.es)

Status: testing

Manufacturer	Device	Name
ADLINK	ACL-7225b	acl7225b
ICP	P16R16DIO	p16r16dio

5.6.3 adl_pci6208 -- ADLINK PCI-6216V

Author: nsyeow <nsyeow@pd.jaring.my>

Status: untested

Manufacturer	Device	Name
ADLINK	PCI-6216V	adl_pci6208

```
Configuration Options:
  none

The driver should work for PCI-6208V, PCI-6208A and PCI-6216V, but all
devices will be treated as a PCI-6216V.

For PCI-6208V and PCI-6208A, only AO channels 0 to 7 are connected and
AO channels 8 to 15 will behave as "phantom" outputs.

The current output ranges for PCI-6208A are not supported.  Only Comedi
sample values 0x8000 to 0xffff should be written to the AO channels on a
PCI-6208A.  Its voltage to current daughter board (EXP-8A) only supports
an input range of 0 to 10 volts and negative voltages may damage the
board.  Comedi sample values 0x0000 to 0x7fff would produce negative
voltages from -10 to 0 volts.
```

5.6.4 adl_pci7230 -- Driver for the ADLINK PCI-7230 32 ch. isolated digital io board

Author: David Fernandez <dfcastelao@gmail.com>

Status: experimental

Manufacturer	Device	Name
ADLINK	PCI-7230	adl_pci7230

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

5.6.5 adl_pci7250 -- Driver for the ADLINK PCI-7250 relay output & digital input card

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

Status: works

Manufacturer	Device	Name
ADLINK	LPCI-7250 LPCIe-7250	adl_pci7250

```
The driver assumes that 3 PCI-7251 modules are fitted to the PCI-7250,
giving 32 channels of relay outputs and 32 channels of isolated digital
inputs.  That is also the case for the LPCI-7250 and LPCIe-7250 cards
although they do not physically support the PCI-7251 modules.

Not fitting the PCI-7251 modules shouldn't do any harm, but the extra
inputs and relay outputs won't work!

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

5.6.6 adl_pci7296 -- Driver for the ADLINK PCI-7296 96 ch. digital io board

Author: Jon Grierson <jd@renko.co.uk>

Status: testing

Manufacturer	Device	Name
ADLINK	PCI-7296	adl_pci7296

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

5.6.7 adl_pci7432 -- Driver for the ADLINK PCI-7432 64 ch. isolated digital io board

Author: Michel Lachaine <mike@mikelachaine.ca>

Status: experimental

Manufacturer	Device	Name
ADLINK	PCI-7432	adl_pci7432

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

5.6.8 adl_pci8164 -- Driver for the ADLINK PCI-8164 4 Axes Motion Control board

Author: Michel Lachaine <mike@mikelachaine.ca>

Status: experimental

Manufacturer	Device	Name
ADLINK	PCI-8164	adl_pci8164

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

5.6.9 adl_pci9111 -- ADLINK PCI-9111HR

Author: Emmanuel Pacaud <emmanuel.pacaud@univ-poitiers.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.
```

5.6.10 adl_pci9112 -- ADLINK PCI-9112

Author: Pascal Berthou <berthou@laas.fr> Emmanuel Pacaud <emmanuel.pacaud@univ-poitiers.fr>

Status: experimental

Manufacturer	Device	Name
ADLINK	PCI-9112	adl_pci9112

```
  - ai_insn read
```

```
  - ao_insn read/write
  - di_insn read
  - do_insn read/write

  Following command mode are not tested
  - 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.
```

5.6.11 adl_pci9118 -- ADLINK PCI-9118DG, PCI-9118HG, PCI-9118HR

Author: Michal Dobes <dobes@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
```

5.6.12 adq12b -- driver for MicroAxial ADQ12-B data acquisition and control card

Author: jeremy theler <thelerg@ib.cnea.gov.ar>

Status: works

Manufacturer	Device	Name
MicroAxial	ADQ12-B	adq12b

```
Driver for the acquisition card ADQ12-B (without any add-on).

 - Analog input is subdevice 0 (16 channels single-ended or 8 differential)
 - Digital input is subdevice 1 (5 channels)
 - Digital output is subdevice 1 (8 channels)
 - The PACER is not supported in this version

If you do not specify any options, they will default to

  # comedi_config /dev/comedi0 adq12b 0x300,0,0

  option 1: I/O base address. The following table is provided as a help
   of the hardware jumpers.

          address            jumper JADR
           0x300                 1 (factory default)
           0x320                 2
```

```
          0x340                 3
          0x360                 4
          0x380                 5
          0x3A0                 6

  option 2: unipolar/bipolar ADC selection: 0 -> bipolar, 1 -> unipolar

        selection         comedi_config option            JUB
         bipolar                0                         2-3 (factory default)
         unipolar               1                         1-2

  option 3: single-ended/differential AI selection: 0 -> SE, 1 -> differential

        selection         comedi_config option     JCHA    JCHB
       single-ended             0                  1-2     1-2 (factory default)
       differential             1                  2-3     2-3


   written by jeremy theler <thelerg@ib.cnea.gov.ar>

   instituto balseiro
   comision nacional de energia atomica
   universidad nacional de cuyo
   argentina

   21-feb-2008
     + changed supported devices string (missused the [] and ())

   13-oct-2007
     + first try
```

5.6.13 adv_pci1710 -- Advantech PCI-1710, PCI-1710HG, PCI-1711, PCI-1713, Advantech PCI-1720, PCI-1731

Author: Michal Dobes <dobes@tesnet.cz>

Status: works

Manufacturer	Device	Name
Advantech	PCI-1710	pci1710 or adv_pci1710
Advantech	PCI-1710HG	pci1710hg
Advantech	PCI-1711	pci1711 or adv_pci1710
Advantech	PCI-1713	pci1713 or adv_pci1710
Advantech	PCI-1716	pci1716 or adv_pci1710
Advantech	PCI-1720	pci1720 or adv_pci1710
Advantech	PCI-1731	pci1731 or adv_pci1710

```
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.
```

5.6.14 adv_pci1723 -- Advantech PCI-1723

Author: yonggang <rsmgnu@gmail.com>, Ian Abbott <abbotti@mev.co.uk>

Status: works

Manufacturer	Device	Name
Advantech	PCI-1723	adv_pci1723

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

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

Subdevice 0 is 8-channel AO, 16-bit, range +/- 10 V.

Subdevice 1 is 16-channel DIO.  The channels are configurable as input or
output in 2 groups (0 to 7, 8 to 15).  Configuring any channel implicitly
configures all channels in the same group.


1. Add the two milliamp ranges to the AO subdevice (0 to 20 mA, 4 to 20 mA).
2. Read the initial ranges and values of the AO subdevice at start-up instead
   of reinitializing them.
3. Implement calibration.
```

5.6.15 adv_pci_dio -- Advantech PCI-1730, PCI-1733, PCI-1734, PCI-1735U, PCI-1736UP, PCI-1750, PCI-1751, PCI-1752, PCI-1753/E, PCI-1754, PCI-1756, PCI-1762

Author: Michal Dobes <dobes@tesnet.cz>

Status: untested

Manufacturer	Device	Name
Advantech	PCI-1730	adv_pci_dio
Advantech	PCI-1733	adv_pci_dio
Advantech	PCI-1734	adv_pci_dio
Advantech	PCI-1735U	adv_pci_dio

Manufacturer	Device	Name
Advantech	PCI-1736UP	adv_pci_dio
Advantech	PCI-1739U	adv_pci_dio
Advantech	PCI-1750	adv_pci_dio
Advantech	PCI-1751	adv_pci_dio
Advantech	PCI-1752	adv_pci_dio
Advantech	PCI-1753	adv_pci_dio
Advantech	PCI-1753+PCI-1753E	adv_pci_dio
Advantech	PCI-1754	adv_pci_dio
Advantech	PCI-1756	adv_pci_dio
Advantech	PCI-1760	adv_pci_dio
Advantech	PCI-1762	adv_pci_dio

```
This driver supports now only insn interface for DI/DO/DIO.

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.
```

5.6.16 aio_aio12_8 -- Acces I/O Products PC-104 AIO12-8 Analog I/O Board

Author: Pablo Mejia <pablo.mejia@cctechnol.com>

Status: experimental

Manufacturer	Device	Name
Acces I/O	PC-104 AIO12-8	adv_pci_dio

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


  Only synchronous operations are supported.
```

5.6.17 aio_iiro_16 -- Acces I/O Products PC-104 IIRO16 Relay And Isolated Input Board

Author: Zachary Ware <zach.ware@cctechnol.com>

Status: experimental

Manufacturer	Device	Name
Acces I/O	PC-104 AIO12-8	adv_pci_dio

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

5.6.18 amplc_dio200 -- Amplicon 200 Series Digital I/O

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

Status: works

Manufacturer	Device	Name
Amplicon	PC212E	pc212e
Amplicon	PC214E	pc214e
Amplicon	PC215E	pc215e
Amplicon	PCI215	pci215 or amplc_dio200
Amplicon	PCIe215	pcie215 or amplc_dio200
Amplicon	PC218E	pc218e
Amplicon	PCIe236	pcie236 or amplc_dio200
Amplicon	PC272E	pc272e
Amplicon	PCI272	pci272 or amplc_dio200
Amplicon	PCIe296	pcie296 or amplc_dio200

```
Configuration options - PC212E, PC214E, PC215E, PC218E, PC272E:
  [0] - I/O port base address
  [1] - IRQ (optional, but commands won't work without it)

Configuration options - PCI215, PCIe215, PCIe236, PCI272, PCIe296:
  [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.

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

SUBDEVICES

                    PC212E         PC214E      PC215E/PCI215
                ------------  -------------  -------------
  Subdevices           6              4              5
   0                PPI-X          PPI-X          PPI-X
   1                CTR-Y1         PPI-Y          PPI-Y
   2                CTR-Y2         CTR-Z1*        CTR-Z1
   3                CTR-Z1       INTERRUPT*       CTR-Z2
   4                CTR-Z2                      INTERRUPT
   5             INTERRUPT

                    PCIe215        PC218E         PCIe236
                ------------  -------------  -------------
  Subdevices           8              7              8
   0                PPI-X          CTR-X1         PPI-X
   1                UNUSED         CTR-X2         UNUSED
```

```
   2                PPI-Y          CTR-Y1          UNUSED
   3                UNUSED         CTR-Y2          UNUSED
   4                CTR-Z1         CTR-Z1          CTR-Z1
   5                CTR-Z2         CTR-Z2          CTR-Z2
   6                TIMER        INTERRUPT         TIMER
   7              INTERRUPT                       INTERRUPT

               PC272E/PCI272     PCIe296
               -------------  -------------
  Subdevices         4              8
   0                PPI-X          PPI-X1
   1                PPI-Y          PPI-X2
   2                PPI-Z          PPI-Y1
   3              INTERRUPT        PPI-Y2
   4                               CTR-Z1
   5                               CTR-Z2
   6                               TIMER
   7                             INTERRUPT
```

Each PPI is a 8255 chip providing 24 DIO channels. The DIO channels are configurable as inputs or outputs in four groups:

```
  Port A  - channels  0 to  7
  Port B  - channels  8 to 15
  Port CL - channels 16 to 19
  Port CH - channels 20 to 23
```

Only mode 0 of the 8255 chips is supported.

Each CTR is a 8254 chip providing 3 16-bit counter channels. Each channel is configured individually with INSN_CONFIG instructions. The specific type of configuration instruction is specified in data[0]. Some configuration instructions expect an additional parameter in data[1]; others return a value in data[1]. The following configuration instructions are supported:

INSN_CONFIG_SET_COUNTER_MODE. Sets the counter channel's mode and BCD/binary setting specified in data[1].

INSN_CONFIG_8254_READ_STATUS. Reads the status register value for the counter channel into data[1].

INSN_CONFIG_SET_CLOCK_SRC. Sets the counter channel's clock source as specified in data[1] (this is a hardware-specific value). Not supported on PC214E. For the other boards, valid clock sources are 0 to 7 as follows:

0. CLK n, the counter channel's dedicated CLK input from the SK1 connector. (N.B. for other values, the counter channel's CLKn pin on the SK1 connector is an output!)
1. Internal 10 MHz clock.
2. Internal 1 MHz clock.
3. Internal 100 kHz clock.
4. Internal 10 kHz clock.
5. Internal 1 kHz clock.
6. OUT n-1, the output of counter channel n-1 (see note 1 below).
7. Ext Clock, the counter chip's dedicated Ext Clock input from the SK1 connector. This pin is shared by all three counter channels on the chip.

INSN_CONFIG_GET_CLOCK_SRC. Returns the counter channel's current clock source in data[1]. For internal clock sources, data[2] is set

```
    to the period in ns.

  INSN_CONFIG_SET_GATE_SRC.  Sets the counter channel's gate source as
    specified in data[2] (this is a hardware-specific value).  Not
    supported on PC214E.  For the other boards, valid gate sources are 0
    to 7 as follows:

      0.  VCC (internal +5V d.c.), i.e. gate permanently enabled.
      1.  GND (internal 0V d.c.), i.e. gate permanently disabled.
      2.  GAT n, the counter channel's dedicated GAT input from the SK1
        connector.  (N.B. for other values, the counter channel's GATn
        pin on the SK1 connector is an output!)
      3.  /OUT n-2, the inverted output of counter channel n-2 (see note
        2 below).
      4.  Reserved.
      5.  Reserved.
      6.  Reserved.
      7.  Reserved.

  INSN_CONFIG_GET_GATE_SRC.  Returns the counter channel's current gate
    source in data[2].

Clock and gate interconnection notes:

  1.  Clock source OUT n-1 is the output of the preceding channel on the
  same counter subdevice if n > 0, or the output of channel 2 on the
  preceding counter subdevice (see note 3) if n = 0.

  2.  Gate source /OUT n-2 is the inverted output of channel 0 on the
  same counter subdevice if n = 2, or the inverted output of channel n+1
  on the preceding counter subdevice (see note 3) if n < 2.

  3.  The counter subdevices are connected in a ring, so the highest
  counter subdevice precedes the lowest.

The 'TIMER' subdevice is a free-running 32-bit timer subdevice.

The 'INTERRUPT' subdevice pretends to be a digital input subdevice.  The
digital inputs come from the interrupt status register.  The number of
channels matches the number of interrupt sources.  The PC214E does not
have an interrupt status register; see notes on 'INTERRUPT SOURCES'
below.

INTERRUPT SOURCES

                   PC212E         PC214E      PC215E/PCI215
                -------------  -------------  -------------
  Sources            6              1              6
   0              PPI-X-C0       JUMPER-J5      PPI-X-C0
   1              PPI-X-C3                      PPI-X-C3
   2             CTR-Y1-OUT1                    PPI-Y-C0
   3             CTR-Y2-OUT1                    PPI-Y-C3
   4             CTR-Z1-OUT1                   CTR-Z1-OUT1
   5             CTR-Z2-OUT1                   CTR-Z2-OUT1

                   PCIe215        PC218E         PCIe236
                -------------  -------------  -------------
  Sources            6              6              6
   0              PPI-X-C0      CTR-X1-OUT1     PPI-X-C0
   1              PPI-X-C3      CTR-X2-OUT1     PPI-X-C3
   2              PPI-Y-C0      CTR-Y1-OUT1      unused
   3              PPI-Y-C3      CTR-Y2-OUT1      unused
```

```
   4              CTR-Z1-OUT1    CTR-Z1-OUT1    CTR-Z1-OUT1
   5              CTR-Z2-OUT1    CTR-Z2-OUT1    CTR-Z2-OUT1

                 PC272E/PCI272     PCIe296
                 -------------  -------------
  Sources              6              6
   0                PPI-X-C0       PPI-X1-C0
   1                PPI-X-C3       PPI-X1-C3
   2                PPI-Y-C0       PPI-Y1-C0
   3                PPI-Y-C3       PPI-Y1-C3
   4                PPI-Z-C0      CTR-Z1-OUT1
   5                PPI-Z-C3      CTR-Z2-OUT1

When an interrupt source is enabled in the interrupt source enable
register, a rising edge on the source signal latches the corresponding
bit to 1 in the interrupt status register.

When the interrupt status register value as a whole (actually, just the
6 least significant bits) goes from zero to non-zero, the board will
generate an interrupt.  For level-triggered hardware interrupts (PCI
card), the interrupt will remain asserted until the interrupt status
register is cleared to zero.  For edge-triggered hardware interrupts
(ISA card), no further interrupts will occur until the interrupt status
register is cleared to zero.  To clear a bit to zero in the interrupt
status register, the corresponding interrupt source must be disabled
in the interrupt source enable register (there is no separate interrupt
clear register).

The PC214E does not have an interrupt source enable register or an
interrupt status register; its 'INTERRUPT' subdevice has a single
channel and its interrupt source is selected by the position of jumper
J5.

COMMANDS

The driver supports a read streaming acquisition command on the
'INTERRUPT' subdevice.  The channel list selects the interrupt sources
to be enabled.  All channels will be sampled together (convert_src ==
TRIG_NOW).  The scan begins a short time after the hardware interrupt
occurs, subject to interrupt latencies (scan_begin_src == TRIG_EXT,
scan_begin_arg == 0).  The value read from the interrupt status register
is packed into a sampl_t value, one bit per requested channel, in the
order they appear in the channel list.
```

5.6.19 amplc_pc236 -- Amplicon PC36AT, PCI236

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

Status: works

Manufacturer	Device	Name
Amplicon	PC36AT	pc36at
Amplicon	PCI236	pci236 or amplc_pc236

```
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 3 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.
```

5.6.20 amplc_pc263 -- Amplicon PC263, PCI263

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

Status: works

Manufacturer	Device	Name
Amplicon	PC263	pc263
Amplicon	PCI263	pci263 or amplc_pc263

```
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.
```

5.6.21 amplc_pci224 -- Amplicon PCI224, PCI234

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

Status: works, but see caveats

Manufacturer	Device	Name
Amplicon	PCI224	amplc_pci224 or pci224
Amplicon	PCI234	amplc_pci224 or pci234

```
  - ao_insn read/write
  - ao_do_cmd mode with the following sources:

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

    The channel list must contain at least one channel with no repeated
    channels.  The scan end count must equal the number of channels in
    the channel list.

    There is only one external trigger source so only one of start_src,
    scan_begin_src or stop_src may use TRIG_EXT.

Configuration options - PCI224:
  [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.
  [2] - Select available ranges according to jumper LK1.  All channels
        are set to the same range:
        0=Jumper position 1-2 (factory default), 4 software-selectable
          internal voltage references, giving 4 bipolar and 4 unipolar
          ranges:
            [-10V,+10V], [-5V,+5V], [-2.5V,+2.5V], [-1.25V,+1.25V],
            [0,+10V], [0,+5V], [0,+2.5V], [0,1.25V].
        1=Jumper position 2-3, 1 external voltage reference, giving
          1 bipolar and 1 unipolar range:
            [-Vext,+Vext], [0,+Vext].

Configuration options - PCI234:
  [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.
  [2] - Select internal or external voltage reference according to
        jumper LK1.  This affects all channels:
        0=Jumper position 1-2 (factory default), Vref=5V internal.
        1=Jumper position 2-3, Vref=Vext external.
  [3] - Select channel 0 range according to jumper LK2:
        0=Jumper position 2-3 (factory default), range [-2*Vref,+2*Vref]
          (10V bipolar when options[2]=0).
        1=Jumper position 1-2, range [-Vref,+Vref]
          (5V bipolar when options[2]=0).
  [4] - Select channel 1 range according to jumper LK3: cf. options[3].
  [5] - Select channel 2 range according to jumper LK4: cf. options[3].
  [6] - Select channel 3 range according to jumper LK5: cf. options[3].

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


  1) All channels on the PCI224 share the same range.  Any change to the
     range as a result of insn_write or a streaming command will affect
```

```
     the output voltages of all channels, including those not specified
     by the instruction or command.

  2) For the analog output command,  the first scan may be triggered
     falsely at the start of acquisition.  This occurs when the DAC scan
     trigger source is switched from 'none' to 'timer' (scan_begin_src =
     TRIG_TIMER) or 'external' (scan_begin_src == TRIG_EXT) at the start
     of acquisition and the trigger source is at logic level 1 at the
     time of the switch.  This is very likely for TRIG_TIMER.  For
     TRIG_EXT, it depends on the state of the external line and whether
     the CR_INVERT flag has been set.  The remaining scans are triggered
     correctly.
```

5.6.22 amplc_pci230 -- Amplicon PCI230, PCI260 Multifunction I/O boards

Author: Allan Willcox <allanwillcox@ozemail.com.au>, Steve D Sharples <steve.sharples@nottingham.ac.uk>, Ian Abbott <abbotti@mev.co.uk>

Status: works

Manufacturer	Device	Name
Amplicon	PCI230	pci230 or amplc_pci230
Amplicon	PCI230+	pci230+ or amplc_pci230
Amplicon	PCI260	pci260 or amplc_pci230
Amplicon	PCI260+	pci260+ or amplc_pci230

```
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.

Configuring a "amplc_pci230" will match any supported card and it will
choose the best match, picking the "+" models if possible.  Configuring
a "pci230" will match a PCI230 or PCI230+ card and it will be treated as
a PCI230.  Configuring a "pci260" will match a PCI260 or PCI260+ card
and it will be treated as a PCI260.  Configuring a "pci230+" will match
a PCI230+ card.  Configuring a "pci260+" will match a PCI260+ card.


                PCI230(+)    PCI260(+)
                ---------    ---------
  Subdevices       3            1
        0          AI           AI
  1          AO
  2          DIO

AI Subdevice:

  The AI subdevice has 16 single-ended channels or 8 differential
  channels.

  The PCI230 and PCI260 cards have 12-bit resolution.  The PCI230+ and
  PCI260+ cards have 16-bit resolution.
```

```
  For differential mode, use inputs 2N and 2N+1 for channel N (e.g. use
  inputs 14 and 15 for channel 7).  If the card is physically a PCI230
  or PCI260 then it actually uses a "pseudo-differential" mode where the
  inputs are sampled a few microseconds apart.  The PCI230+ and PCI260+
  use true differential sampling.  Another difference is that if the
  card is physically a PCI230 or PCI260, the inverting input is 2N,
  whereas for a PCI230+ or PCI260+ the inverting input is 2N+1.  So if a
  PCI230 is physically replaced by a PCI230+ (or a PCI260 with a
  PCI260+) and differential mode is used, the differential inputs need
  to be physically swapped on the connector.

  The following input ranges are supported:

    0 => [-10, +10] V
    1 => [-5, +5] V
    2 => [-2.5, +2.5] V
    3 => [-1.25, +1.25] V
    4 => [0, 10] V
    5 => [0, 5] V
    6 => [0, 2.5] V

AI Commands:

  +=========+==============+===========+============+==========+
  |start_src|scan_begin_src|convert_src|scan_end_src| stop_src |
  +=========+==============+===========+============+==========+
  |TRIG_NOW | TRIG_FOLLOW  |TRIG_TIMER | TRIG_COUNT |TRIG_NONE |
  |TRIG_INT |              |TRIG_EXT(3)|            |TRIG_COUNT|
  |         |              |TRIG_INT   |            |          |
  |         |--------------|-----------|            |          |
  |         | TRIG_TIMER(1)|TRIG_TIMER |            |          |
  |         | TRIG_EXT(2)  |           |            |          |
  |         | TRIG_INT     |           |            |          |
  +---------+--------------+-----------+------------+----------+

  Note 1: If AI command and AO command are used simultaneously, only
          one may have scan_begin_src == TRIG_TIMER.

  Note 2: For PCI230 and PCI230+, scan_begin_src == TRIG_EXT uses
          DIO channel 16 (pin 49) which will need to be configured as
          a digital input.  For PCI260+, the EXTTRIG/EXTCONVCLK input
          (pin 17) is used instead.  For PCI230, scan_begin_src ==
          TRIG_EXT is not supported.  The trigger is a rising edge
          on the input.

  Note 3: For convert_src == TRIG_EXT, the EXTTRIG/EXTCONVCLK input
          (pin 25 on PCI230(+), pin 17 on PCI260(+)) is used.  The
          convert_arg value is interpreted as follows:

            convert_arg == (CR_EDGE | 0) => rising edge
            convert_arg == (CR_EDGE | CR_INVERT | 0) => falling edge
            convert_arg == 0 => falling edge (backwards compatibility)
            convert_arg == 1 => rising edge (backwards compatibility)

  All entries in the channel list must use the same analogue reference.
  If the analogue reference is not AREF_DIFF (not differential) each
  pair of channel numbers (0 and 1, 2 and 3, etc.) must use the same
  input range.  The input ranges used in the sequence must be all
  bipolar (ranges 0 to 3) or all unipolar (ranges 4 to 6).  The channel
  sequence must consist of 1 or more identical subsequences.  Within the
  subsequence, channels must be in ascending order with no repeated
```

```
  channels.  For example, the following sequences are valid: 0 1 2 3
  (single valid subsequence), 0 2 3 5 0 2 3 5 (repeated valid
  subsequence), 1 1 1 1 (repeated valid subsequence).  The following
  sequences are invalid: 0 3 2 1 (invalid subsequence), 0 2 3 5 0 2 3
  (incompletely repeated subsequence).  Some versions of the PCI230+ and
  PCI260+ have a bug that requires a subsequence longer than one entry
  long to include channel 0.

AO Subdevice:

  The AO subdevice has 2 channels with 12-bit resolution.

  The following output ranges are supported:

    0 => [0, 10] V
    1 => [-10, +10] V

AO Commands:

  +=========+==============+===========+============+==========+
  |start_src|scan_begin_src|convert_src|scan_end_src| stop_src |
  +=========+==============+===========+============+==========+
  |TRIG_INT | TRIG_TIMER(1)| TRIG_NOW  | TRIG_COUNT |TRIG_NONE |
  |         | TRIG_EXT(2)  |           |            |TRIG_COUNT|
  |         | TRIG_INT     |           |            |          |
  +---------+--------------+-----------+------------+----------+

  Note 1: If AI command and AO command are used simultaneously, only
          one may have scan_begin_src == TRIG_TIMER.

  Note 2: scan_begin_src == TRIG_EXT is only supported if the card is
          configured as a PCI230+ and is only supported on later
          versions of the card.  As a card configured as a PCI230+ is
          not guaranteed to support external triggering, please consider
          this support to be a bonus.  It uses the EXTTRIG/ EXTCONVCLK
          input (PCI230+ pin 25).  Triggering will be on the rising edge
          unless the CR_INVERT flag is set in scan_begin_arg.

  The channels in the channel sequence must be in ascending order with
  no repeats.  All entries in the channel sequence must use the same
  output range.

DIO Subdevice:

  The DIO subdevice is a 8255 chip providing 24 DIO channels.  The DIO
  channels are configurable as inputs or outputs in four groups:

    Port A  - channels  0 to  7
    Port B  - channels  8 to 15
    Port CL - channels 16 to 19
    Port CH - channels 20 to 23

  Only mode 0 of the 8255 chip is supported.

  Bit 0 of port C (DIO channel 16) is also used as an external scan
  trigger input for AI commands on PCI230 and PCI230+, so would need to
  be configured as an input to use it for that purpose.
```

5.6.23 c6xdigio -- Mechatronic Systems Inc. C6x_DIGIO DSP daughter card

Author: Dan Block

Status: unknown

Manufacturer	Device	Name
Mechatronic Systems Inc.	C6x_DIGIO DSP daughter card	c6xdigio

```
This driver will not work with a 2.4 kernel.
```

5.6.24 cb_das16_cs -- Computer Boards PC-CARD DAS16/16

Author: ds

Status: experimental

Manufacturer	Device	Name
ComputerBoards	PC-CARD DAS16/16	cb_das16_cs
ComputerBoards	PC-CARD DAS16/16-AO	cb_das16_cs

5.6.25 cb_pcidac -- Measurement Computing PCI Migration series boards

Author: Oliver Gause

Status: works

Manufacturer	Device	Name
ComputerBoards	PCI-DAC6702	cb_pcidac
ComputerBoards	PCI-DAC6703	cb_pcidac

```
Written to support the PCI-DAC6702. Trivially extended to support
the PCI-DAC6703, it has just 16 ao channels instead of 8.

Configuration Options:
    [0] - PCI bus number
    [1] - PCI slot number

Developed from cb_pcidas64, cb_pcimdas and skel. The register values are
taken from the register map of Measurement Computing.
```

```
Supports DIO, AO in its present form.
```

5.6.26 cb_pcidas -- MeasurementComputing PCI-DAS series with the AMCC S5933 PCI controller

Author: Ivan Martinez <imr@oersted.dtu.dk>, Frank Mori Hess <fmhess@users.sourceforge.net>, Brice Dubost <braice@braice.net>

Status: There are many reports of the driver being used with most of the supported cards. Despite no detailed log is maintained, it can be said that the driver is quite tested and stable.

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 may be autocalibrated using the comedi_calibrate
  utility.

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

AI Triggering:
   For start_src == TRIG_EXT, the A/D EXTERNAL TRIGGER IN (pin 45) is used.
   For 1602 series, the start_arg is interpreted as follows:
     start_arg == 0                  => gated triger (level high)
     start_arg == CR_INVERT          => gated triger (level low)
     start_arg == CR_EDGE            => Rising edge
     start_arg == CR_EDGE | CR_INVERT => Falling edge
   For the other boards the trigger will be done on rising edge
```

5.6.27 cb_pcidas64 -- MeasurementComputing PCI-DAS64xx, 60XX, and 4020 series with the PLX 9080 PCI controller

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

Status: works

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-DAS6013	cb_pcidas64
Measurement Computing	PCI-DAS6014	cb_pcidas64
Measurement Computing	PCI-DAS6023	cb_pcidas64
Measurement Computing	PCI-DAS6025	cb_pcidas64
Measurement Computing	PCI-DAS6030	cb_pcidas64
Measurement Computing	PCI-DAS6031	cb_pcidas64
Measurement Computing	PCI-DAS6032	cb_pcidas64
Measurement Computing	PCI-DAS6033	cb_pcidas64
Measurement Computing	PCI-DAS6034	cb_pcidas64
Measurement Computing	PCI-DAS6035	cb_pcidas64
Measurement Computing	PCI-DAS6036	cb_pcidas64
Measurement Computing	PCI-DAS6040	cb_pcidas64
Measurement Computing	PCI-DAS6052	cb_pcidas64
Measurement Computing	PCI-DAS6070	cb_pcidas64
Measurement Computing	PCI-DAS6071	cb_pcidas64
Measurement Computing	PCI-DAS4020/12	cb_pcidas64

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

These boards may be autocalibrated with the comedi_calibrate utility.

To select the bnc trigger input on the 4020 (instead of the dio input),
specify a nonzero channel in the chanspec.  If you wish to use an external
master clock on the 4020, you may do so by setting the scan_begin_src
to TRIG_OTHER, and using an INSN_CONFIG_TIMER_1 configuration insn
to configure the divisor to use for the external clock.

Some devices are not identified because the PCI device IDs are not yet
known. If you have such a board, please file a bug report at
```

5.6.28 cb_pcidda -- 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 success/failure with other different cards to
<comedi@comedi.org>.
```

5.6.29 cb_pcidio -- ComputerBoards' DIO boards with PCI interface

Author: Yoshiya Matsuzaka

Status: experimental

Manufacturer	Device	Name
Measurement Computing	PCI-DIO24	cb_pcidio
Measurement Computing	PCI-DIO24H	cb_pcidio
Measurement Computing	PCI-DIO48H	cb_pcidio

```
This driver has been modified from skel.c of comedi-0.7.70.

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.

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

5.6.30 cb_pcimdas -- Measurement Computing PCI Migration series boards

Author: Richard Bytheway

Status: experimental

Manufacturer	Device	Name
ComputerBoards	PCIM-DAS1602/16	cb_pcimdas
ComputerBoards	PCIe-DAS1602/16	cb_pcimdas

```
Written to support the PCIM-DAS1602/16 and PCIe-DAS1602/16.

Configuration Options:
    [0] - PCI bus number
    [1] - PCI slot number

Developed from cb_pcidas and skel by Richard Bytheway (mocelet@sucs.org).
Only supports DIO, AO and simple AI in it's present form.
No interrupts, multi channel or FIFO AI, although the card looks like it could
support this.
```

5.6.31 cb_pcimdda -- Measurement Computing PCIM-DDA06-16

Author: Calin Culianu <calin@ajvar.org>

Status: works

Manufacturer	Device	Name
Measurement Computing	PCIM-DDA06-16	cb_pcimdda

```
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:
  [0] PCI bus (optional)
  [1] PCI slot (optional)
  [2] analog output range jumper setting
      0 == +/- 5 V
      1 == +/- 10 V
```

5.6.32 comedi_bond -- A driver to 'bond' (merge) multiple subdevices from multiple devices together as one.

Author: ds

Status: works

```
This driver allows you to 'bond' (merge) multiple comedi subdevices
(coming from possibly difference boards and/or drivers) together.  For
example, if you had a board with 2 different DIO subdevices, and
another with 1 DIO subdevice, you could 'bond' them with this driver
so that they look like one big fat DIO subdevice.  This makes writing
applications slightly easier as you don't have to worry about managing
different subdevices in the application -- you just worry about
indexing one linear array of channel id's.

Right now only DIO subdevices are supported as that's the personal itch
I am scratching with this driver.  If you want to add support for AI and AO
subdevs, go right on ahead and do so!

Commands aren't supported -- although it would be cool if they were.

Configuration Options:
  List of comedi-minors to bond.  All subdevices of the same type
  within each minor will be concatenated together in the order given here.
```

5.6.33 comedi_parport -- 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.
```

5.6.34 comedi_rt_timer -- 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
```

5.6.35 comedi_test -- generates fake waveforms

Author: Joachim Wuttke <Joachim.Wuttke@icn.siemens.de>, Frank Mori Hess <fmhess@users.sourceforge.net>, 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).
```

5.6.36 contec_fit -- Contec F&eIT series modules

Author: Contec Co., Ltd.

Status: works

Manufacturer	Device	Name
Contec	GY	contec_fit
Contec	GY	FIT
Contec	GY	FIT

```
Configuration Options:
  [0] - DeviceID of module (optional)
  If DeviceID is not specified, DeviceID 0 will be used.
```

5.6.37 contec_pci_dio -- Contec PIO1616L digital I/O board

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

Status: works

Manufacturer	Device	Name
Contec	PIO1616L	contec_pci_dio

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

5.6.38 daqboard2000 -- 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
be loaded by comedi_config using the -i
option.  The initialization code is available from http://www.comedi.org
in the comedi_nonfree_firmware tarball.

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

5.6.39 das08 -- DAS-08 compatible boards

Author: Warren Jasper, ds, Frank Hess

Status: works

Manufacturer	Device	Name
Keithley Metrabyte	DAS08	isa-das08
ComputerBoards	DAS08	isa-das08

Manufacturer	Device	Name
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	das08
ComputerBoards	PC104-DAS08	pc104-das08
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)

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

5.6.40 das08_cs -- DAS-08 PCMCIA boards

Author: Warren Jasper, ds, Frank Hess

Status: works

Manufacturer	Device	Name
ComputerBoards	PCM-DAS08	pcm-das08

```
This is the PCMCIA-specific support split off from the
das08 driver.

Options (for pcm-das08):
        NONE

Command support does not exist, but could be added for this board.
```

5.6.41 das16 -- 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
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.
```

5.6.42 das16m1 -- CIO-DAS16/M1

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

Status: works

Manufacturer	Device	Name
Measurement Computing	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.
```

5.6.43 das1800 -- Keithley Metrabyte DAS1800 (& compatibles)

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

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)
```

5.6.44 das6402 -- Keithley Metrabyte DAS6402 (& compatibles)

Author: Oystein Svendsen <svendsen@pvv.org>

Status: bitrotten

Manufacturer	Device	Name
Keithley Metrabyte	DAS6402	das6402

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

5.6.45 das800 -- Keithley Metrabyte DAS800 (& compatibles)

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

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.
```

5.6.46 dmm32at -- Diamond Systems mm32at driver.

Author: Perry J. Piplani <perry.j.piplani@nasa.gov>

Status: experimental

```
This driver is for the Diamond Systems MM-32-AT board

Configuration Options:
  comedi_config /dev/comedi0 dmm32at baseaddr,irq
```

5.6.47 dt2801 -- 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)
```

5.6.48 dt2811 -- Data Translation DT2811

Author: ds

Status: works

Manufacturer	Device	Name
Data Translation	DT2811-PGL	dt2811-pgl

Manufacturer	Device	Name
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 = single-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)
```

5.6.49 dt2814 -- 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.
```

5.6.50 dt2815 -- 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)
```

5.6.51 dt2817 -- 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
```

5.6.52 dt282x -- Data Translation DT2821 series (including DT-EZ)

Author: ds

Status: complete

Manufacturer	Device	Name
Data Translation	DT2821	dt2821
Data Translation	DT2821-F-16SE	dt2821-f
Data Translation	DT2821-F-8DI	dt2821-f
Data Translation	DT2821-G-16SE	dt2821-f
Data Translation	DT2821-G-8DI	dt2821-g
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]
```

5.6.53 dt3000 -- 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

Manufacturer	Device	Name
Data Translation	DT3005	dt3000
Data Translation	DT3004-200	dt3000

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

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

AO commands are not supported.
```

5.6.54 dt9812 -- Data Translation DT9812 USB module

Author: anders.blomdell@control.lth.se (Anders Blomdell)

Status: in development

Manufacturer	Device	Name
Data Translation	DT9812	dt9812

```
This driver works, but bulk transfers not implemented. Might be a starting point
for someone else. I found out too late that USB has too high latencies (>1 ms)
for my needs.
```

5.6.55 fl512 -- unknown

Author: Anders Gnistrup <ex18@kalman.iau.dtu.dk>

Status: unknown

Manufacturer	Device	Name
unknown	FL512	fl512

```
Digital I/O is not supported.

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

5.6.56 gsc_hpdi -- General Standards Corporation High Speed Parallel Digital Interface rs485 boards

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

Status: only receive mode works, transmit not supported

Manufacturer	Device	Name
General Standards Corporation	PCI-HPDI32	gsc_hpdi
General Standards Corporation	PMC-HPDI32	gsc_hpdi

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

There are some additional hpdi models available from GSC for which
support could be added to this driver.
```

5.6.57 icp_multi -- Inova ICP_MULTI

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

Status: works

Manufacturer	Device	Name
Inova	ICP_MULTI	icp_multi

```
The driver works for analog input and output and digital input and output.
It does not work with interrupts or with the counters.  Currently no support
for DMA.

It has 16 single-ended or 8 differential Analogue Input channels with 12-bit
resolution.  Ranges : 5V, 10V, +/-5V, +/-10V, 0..20mA and 4..20mA.  Input
ranges can be individually programmed for each channel.  Voltage or current
measurement is selected by jumper.

There are 4 x 12-bit Analogue Outputs.  Ranges : 5V, 10V, +/-5V, +/-10V

16 x Digital Inputs, 24V

8 x Digital Outputs, 24V, 1A

4 x 16-bit counters
```

5.6.58 ii_pci20kc -- 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
```

5.6.59 jr3_pci -- JR3/PCI force sensor board

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

Status: works

Manufacturer	Device	Name
JR3	PCI force sensor board	jr3_pci

```
  The DSP on the board requires initialization code, which can
  be loaded by placing it in /lib/firmware/comedi.
  The initialization code should be somewhere on the media you got
  with your card. One version is available from http://www.comedi.org
  in the comedi_nonfree_firmware tarball.

  Configuration options:
  [0] - PCI bus number - if bus number and slot number are 0,
                         then driver search for first unused card
  [1] - PCI slot number
```

5.6.60 ke_counter -- Driver for Kolter Electronic Counter Card

Author: Michael Hillmann

Status: tested

Manufacturer	Device	Name
Kolter Electronic	PCI Counter Card	ke_counter

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

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

5.6.61 me4000 -- Meilhaus ME-4000 series boards

Author: gg (Guenter Gebhardt <g.gebhardt@meilhaus.com>)

Status: broken (no support for loading firmware)

Manufacturer	Device	Name
Meilhaus	ME-4650	me4000
Meilhaus	ME-4670i	me4000
Meilhaus	ME-4680	me4000
Meilhaus	ME-4680i	me4000
Meilhaus	ME-4680is	me4000

```
    - Analog Input
    - Analog Output
    - Digital I/O
    - Counter

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.

The firmware required by these boards is available in the
comedi_nonfree_firmware tarball available from
```

5.6.62 me_daq -- Meilhaus PCI data acquisition cards

Author: Michael Hillmann <hillmann@syscongroup.de>

Status: experimental

Manufacturer	Device	Name
Meilhaus	ME-2600i	me_daq
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.

The 2600 requires a firmware upload, which can be accomplished
using the -i or --init-data option of comedi_config.
The firmware can be
found in the comedi_nonfree_firmware tarball available
from http://www.comedi.org
```

5.6.63 mpc624 -- Micro/sys MPC-624 PC/104 board

Author: Stanislaw Raczynski <sraczynski@op.pl>

Status: working

Manufacturer	Device	Name
Micro/sys	MPC-624	mpc624

```
    The Micro/sys MPC-624 board is based on the LTC2440 24-bit sigma-delta
    ADC chip.

    Subdevices supported by the driver:
    - Analog In:   supported
    - Digital I/O: not supported
    - LEDs:        not supported
    - EEPROM:      not supported

Configuration Options:
  [0] - I/O base address
  [1] - convertion rate
            Convertion rate  RMS noise  Effective Number Of Bits
          0      3.52kHz        23uV               17
```

```
        1      1.76kHz       3.5uV                20
        2       880Hz         2uV               21.3
        3       440Hz        1.4uV              21.8
        4       220Hz         1uV               22.4
        5       110Hz        750uV              22.9
        6       55Hz         510nV              23.4
        7      27.5Hz        375nV               24
        8      13.75Hz       250nV              24.4
        9      6.875Hz       200nV              24.6
  [2] - voltage range
        0      -1.01V .. +1.01V
        1      -10.1V .. +10.1V
```

5.6.64 mpc8260cpm -- 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.
```

5.6.65 multiq3 -- Quanser Consulting MultiQ-3

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

Status: works

Manufacturer	Device	Name
Quanser Consulting	MultiQ-3	multiq3

5.6.66 ni_6527 -- National Instruments 6527

Author: ds

Status: works

Manufacturer	Device	Name
National Instruments	PCI-6527	ni6527
National Instruments	PXI-6527	ni6527

5.6.67 ni_65xx -- National Instruments 65xx static dio boards

Author: Jon Grierson <jd@renko.co.uk>, Frank Mori Hess <fmhess@users.sourceforge.net>

Status: testing

Manufacturer	Device	Name
National Instruments	PCI-6509	ni_65xx
National Instruments	PXI-6509	ni_65xx
National Instruments	PCI-6510	ni_65xx
National Instruments	PCI-6511	ni_65xx
National Instruments	PXI-6511	ni_65xx
National Instruments	PCI-6512	ni_65xx
National Instruments	PXI-6512	ni_65xx
National Instruments	PCI-6513	ni_65xx
National Instruments	PXI-6513	ni_65xx
National Instruments	PCI-6514	ni_65xx
National Instruments	PXI-6514	ni_65xx
National Instruments	PCI-6515	ni_65xx
National Instruments	PXI-6515	ni_65xx
National Instruments	PCI-6516	ni_65xx
National Instruments	PCI-6517	ni_65xx
National Instruments	PCI-6518	ni_65xx
National Instruments	PCI-6519	ni_65xx
National Instruments	PCI-6520	ni_65xx
National Instruments	PCI-6521	ni_65xx
National Instruments	PXI-6521	ni_65xx
National Instruments	PCI-6528	ni_65xx
National Instruments	PXI-6528	ni_65xx

```
Based on the PCI-6527 driver by ds.
The interrupt subdevice (subdevice 3) is probably broken for all boards
except maybe the 6514.
```

5.6.68 ni_660x -- National Instruments 660x counter/timer boards

Author: J.P. Mellor <jpmellor@rose-hulman.edu>, Herman.Bruyninckx@mech.kuleuven.ac.be, Wim.Meeussen@mech.kuleuven.ac.be, Klaas.Gadeyne@mech.kuleuven.ac.be, Frank Mori Hess <fmhess@users.sourceforge.net>

Status: experimental

Manufacturer	Device	Name
National Instruments	PCI-6601	ni_660x
National Instruments	PCI-6602	ni_660x
National Instruments	PXI-6602	ni_660x
National Instruments	PXI-6608	ni_660x
National Instruments	PCI-6624	ni_660x
National Instruments	PXI-6624	ni_660x

```
Encoders work.  PulseGeneration (both single pulse and pulse train)
works. Buffered commands work for input but not output.
```

5.6.69 ni_670x -- 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

```
Commands are not supported.
```

5.6.70 ni_at_a2150 -- 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)
```

5.6.71 ni_at_ao -- National Instruments AT-AO-6/10

Author: ds

Status: should work

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

```
Configuration options:
  [0] - I/O port base address
  [1] - IRQ (unused)
  [2] - DMA (unused)
  [3] - analog output range, set by jumpers on hardware (0 for -10 to 10V bipolar, 1 for 0V ↩
       to 10V unipolar)
```

5.6.72 ni_atmio -- 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
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 driver has 2.6 kernel isapnp support, and
will automatically probe for a supported board if the
I/O base is left unspecified with comedi_config.
However, many of
the isapnp id numbers are unknown.  If your board is not
recognized, please send the output of 'cat /proc/isapnp'
(you may need to modprobe the isa-pnp module for
/proc/isapnp to exist) so the
id numbers for your board can be added to the driver.
```

```
Otherwise, you can use the isapnptools package to configure
your board.  Use isapnp to
configure the I/O base and IRQ for the board, and then pass
the same values as
parameters 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.
```

5.6.73 ni_atmio16d -- 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

5.6.74 ni_daq_700 -- National Instruments PCMCIA DAQCard-700 DIO only

Author: Fred Brooks <nsaspook@nsaspook.com>, based on ni_daq_dio24 by Daniel Vecino Castel <dvecino@able.es>

Status: works

Manufacturer	Device	Name
National Instruments	PCMCIA DAQ-Card-700	ni_daq_700

```
The daqcard-700 appears in Comedi as a single digital I/O subdevice with
16 channels.  The channel 0 corresponds to the daqcard-700's output
port, bit 0; channel 8 corresponds to the input port, bit 0.

Direction configuration: channels 0-7 output, 8-15 input (8225 device
emu as port A output, port B input, port C N/A).
```

```
IRQ is assigned but not used.
```

5.6.75 ni_daq_dio24 -- National Instruments PCMCIA DAQ-Card DIO-24

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

Status: ?

Manufacturer	Device	Name
National Instruments	PCMCIA DAQ-Card DIO-24	ni_daq_dio24

```
This is just a wrapper around the 8255.o driver to properly handle
the PCMCIA interface.
```

5.6.76 ni_labpc -- National Instruments Lab-PC (& compatibles)

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

Status: works

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

```
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).

Kernel-level ISA plug-and-play support for the lab-pc-1200
boards has not
yet been added to the driver, mainly due to the fact that
I don't know the device id numbers.  If you have one
of these boards,
please file a bug report at https://bugs.comedi.org/
so I can get the necessary information from you.

The 1200 series boards have onboard calibration dacs for correcting
analog input/output offsets and gains.  The proper settings for these
caldacs are stored on the board's eeprom.  To read the caldac values
from the eeprom and store them into a file that can be then be used by
comedilib, use the comedi_calibrate program.

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)

The Lab-pc+ has quirky chanlist requirements
when scanning multiple channels.  Multiple channel scan
sequence must start at highest channel, then decrement down to
channel 0.  The rest of the cards can scan down like lab-pc+ or scan
up from channel zero.  Chanlists consisting of all one channel
are also legal, and allow you to pace conversions in bursts.
```

5.6.77 ni_labpc_cs -- National Instruments Lab-PC (& compatibles)

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

Status: works

Manufacturer	Device	Name
National Instruments	DAQCard-1200	daqcard-1200

```
Thanks go to Fredrik Lingvall for much testing and perseverance in
helping to debug daqcard-1200 support.

The 1200 series boards have onboard calibration dacs for correcting
analog input/output offsets and gains.  The proper settings for these
caldacs are stored on the board's eeprom.  To read the caldac values
from the eeprom and store them into a file that can be then be used by
comedilib, use the comedi_calibrate program.

Configuration options:
  none

The daqcard-1200 has quirky chanlist requirements
when scanning multiple channels.  Multiple channel scan
sequence must start at highest channel, then decrement down to
channel 0.  Chanlists consisting of all one channel
are also legal, and allow you to pace conversions in bursts.
```

5.6.78 ni_mio_cs -- 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
National Instruments	DAQCard-6036E	ni_mio_cs

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

5.6.79 ni_pcidio -- 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-6533	ni_pcidio
National Instruments	PCI-6534	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.

The PCI-DIO-32HS/PCI-6533 has a configurable external trigger. Setting
scan_begin_arg to 0 or CR_EDGE triggers on the leading edge. Setting
scan_begin_arg to CR_INVERT or (CR_EDGE | CR_INVERT) triggers on the
trailing edge.

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

The PCI-6534 requires a firmware upload after power-up to work, the
firmware data and instructions for loading it with comedi_config
it are contained in the
comedi_nonfree_firmware tarball available from http://www.comedi.org
```

5.6.80 ni_pcimio -- National Instruments PCI-MIO-E series and M series (all boards)

Author: ds, John Hallen, Frank Mori Hess, Rolf Mueller, Herbert Peremans, Herman Bruyninckx, Terry Barnaby

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-6014	ni_pcimio
National Instruments	PCI-6040E	ni_pcimio
National Instruments	PXI-6040E	ni_pcimio
National Instruments	PCI-6030E	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-6220	ni_pcimio
National Instruments	PXI-6220	ni_pcimio
National Instruments	PCI-6221	ni_pcimio
National Instruments	PXI-6221	ni_pcimio
National Instruments	PCI-6224	ni_pcimio
National Instruments	PXI-6224	ni_pcimio
National Instruments	PCI-6225	ni_pcimio
National Instruments	PXI-6225	ni_pcimio
National Instruments	PCI-6229	ni_pcimio
National Instruments	PCI-6250	ni_pcimio
National Instruments	PXI-6250	ni_pcimio
National Instruments	PCI-6251	ni_pcimio
National Instruments	PXI-6251	ni_pcimio
National Instruments	PCIe-6251	ni_pcimio
National Instruments	PXIe-6251	ni_pcimio
National Instruments	PCI-6254	ni_pcimio
National Instruments	PXI-6254	ni_pcimio
National Instruments	PCI-6259	ni_pcimio
National Instruments	PXI-6259	ni_pcimio
National Instruments	PCIe-6259	ni_pcimio
National Instruments	PXIe-6259	ni_pcimio
National Instruments	PCI-6280	ni_pcimio

Manufacturer	Device	Name
National Instruments	PXI-6280	ni_pcimio
National Instruments	PCI-6281	ni_pcimio
National Instruments	PXI-6281	ni_pcimio
National Instruments	PCI-6284	ni_pcimio
National Instruments	PXI-6284	ni_pcimio
National Instruments	PCI-6289	ni_pcimio
National Instruments	PXI-6289	ni_pcimio
National Instruments	PCI-6711	ni_pcimio
National Instruments	PXI-6711	ni_pcimio
National Instruments	PCI-6713	ni_pcimio
National Instruments	PXI-6713	ni_pcimio
National Instruments	PXI-6071E	ni_pcimio
National Instruments	PCI-6070E	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
National Instruments	PCI-6733	ni_pcimio
National Instruments	PXI-6733	ni_pcimio
National Instruments	PCI-6143	ni_pcimio
National Instruments	PXI-6143	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
comedi_calibrate (or comedi_soft_calibrate for m-series) utility.
M-Series boards do analog input and analog output calibration entirely
in software. The software calibration corrects
the analog input for offset, gain and
nonlinearity.  The analog outputs are corrected for offset and gain.
See the comedilib documentation on comedi_get_softcal_converter() for
more information.

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

Digital I/O may not work on 673x.

Note that the PCI-6143 is a simultaineous sampling device with 8 convertors.
With this board all of the convertors perform one simultaineous sample during
a scan interval. The period for a scan is used for the convert time in a
Comedi cmd. The convert trigger source is normally set to TRIG_NOW by default.

The RTSI trigger bus is supported on these cards on
subdevice 10. See the comedilib documentation for details.

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.

SCXI is probably broken for m-series boards.
```

5.6.81 ni_tio -- National Instruments general purpose counters

Author: J.P. Mellor <jpmellor@rose-hulman.edu>, Herman.Bruyninckx@mech.kuleuven.ac.be, Wim.Meeussen@mech.kuleuven.ac.be, Klaas.Gadeyne@mech.kuleuven.ac.be, Frank Mori Hess <fmhess@users.sourceforge.net>

Status: works

```
This module is not used directly by end-users.  Rather, it
is used by other drivers (for example ni_660x and ni_pcimio)
to provide support for NI's general purpose counters.  It was
originally based on the counter code from ni_660x.c and
ni_mio_common.c.
```

5.6.82 ni_tiocmd -- National Instruments general purpose counters command support

Author: J.P. Mellor <jpmellor@rose-hulman.edu>, Herman.Bruyninckx@mech.kuleuven.ac.be, Wim.Meeussen@mech.kuleuven.ac.be, Klaas.Gadeyne@mech.kuleuven.ac.be, Frank Mori Hess <fmhess@users.sourceforge.net>

Status: works

```
This module is not used directly by end-users.  Rather, it
is used by other drivers (for example ni_660x and ni_pcimio)
to provide command support for NI's general purpose counters.
It was originally split out of ni_tio.c to stop the 'ni_tio'
module depending on the 'mite' module.
```

5.6.83 pcl711 -- 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.
```

5.6.84 pcl724 -- Advantech PCL-724, PCL-722, PCL-731 ADLINK ACL-7122, ACL-7124, PET-48DIO

Author: Michal Dobes <dobes@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.
```

5.6.85 pcl725 -- Advantech PCL-725 (& compatibles)

Author: ds

Status: unknown

Manufacturer	Device	Name
Advantech	PCL-725	pcl725

5.6.86 pcl726 -- Advantech PCL-726 & compatibles

Author: ds

Status: untested

Manufacturer	Device	Name
Advantech	PCL-726	pcl726

Manufacturer	Device	Name
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
```

5.6.87 pcl730 -- Advantech PCL-730 (& compatibles)

Author: José Luis Sánchez (jsanchezv@teleline.es)

Status: untested

Manufacturer	Device	Name
Advantech	PCL-730	pcl730
ICP	ISO-730	iso730
ICP	[ADLINK] ACL-7130	acl7130

```
Interrupts are not supported.
The ACL-7130 card have an 8254 timer/counter not supported by this driver.
```

5.6.88 pcl812 -- 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 <dobes@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
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.

Options for PCL-812:
  [0] - IO Base
  [1] - IRQ  (0=disable, 2, 3, 4, 5, 6, 7; 10, 11, 12, 14, 15)
  [2] - DMA  (0=disable, 1, 3)
  [3] - 0=trigger source is internal 8253 with 2MHz clock
        1=trigger source is external
  [4] - 0=A/D input range is +/-10V
        1=A/D input range is +/-5V
        2=A/D input range is +/-2.5V
        3=A/D input range is +/-1.25V
        4=A/D input range is +/-0.625V
        5=A/D input range is +/-0.3125V
  [5] - 0=D/A outputs 0-5V  (internal reference -5V)
        1=D/A outputs 0-10V (internal reference -10V)
        2=D/A outputs unknow (external reference)

Options for PCL-812PG, ACL-8112PG:
  [0] - IO Base
  [1] - IRQ  (0=disable, 2, 3, 4, 5, 6, 7; 10, 11, 12, 14, 15)
  [2] - DMA  (0=disable, 1, 3)
  [3] - 0=trigger source is internal 8253 with 2MHz clock
        1=trigger source is external
  [4] - 0=A/D have max +/-5V input
        1=A/D have max +/-10V input
  [5] - 0=D/A outputs 0-5V  (internal reference -5V)
        1=D/A outputs 0-10V (internal reference -10V)
        2=D/A outputs unknow (external reference)
```

```
Options for ACL-8112DG/HG, A-822PGL/PGH, A-823PGL/PGH, ACL-8216, A-826PG:
  [0] - IO Base
  [1] - IRQ  (0=disable, 2, 3, 4, 5, 6, 7; 10, 11, 12, 14, 15)
  [2] - DMA  (0=disable, 1, 3)
  [3] - 0=trigger source is internal 8253 with 2MHz clock
        1=trigger source is external
  [4] - 0=A/D channels are S.E.
        1=A/D channels are DIFF
  [5] - 0=D/A outputs 0-5V  (internal reference -5V)
        1=D/A outputs 0-10V (internal reference -10V)
        2=D/A outputs unknow (external reference)

Options for A-821PGL/PGH:
  [0] - IO Base
  [1] - IRQ  (0=disable, 2, 3, 4, 5, 6, 7)
  [2] - 0=A/D channels are S.E.
        1=A/D channels are DIFF
  [3] - 0=D/A output 0-5V  (internal reference -5V)
        1=D/A output 0-10V (internal reference -10V)

Options for A-821PGL-NDA:
  [0] - IO Base
  [1] - IRQ  (0=disable, 2, 3, 4, 5, 6, 7)
  [2] - 0=A/D channels are S.E.
        1=A/D channels are DIFF

Options for PCL-813:
  [0] - IO Base

Options for PCL-813B:
  [0] - IO Base
  [1] - 0= bipolar inputs
        1= unipolar inputs

Options for ACL-8113, ISO-813:
  [0] - IO Base
  [1] - 0= 10V bipolar inputs
        1= 10V unipolar inputs
        2= 20V bipolar inputs
        3= 20V unipolar inputs
```

5.6.89 pcl816 -- 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
```

5.6.90 pcl818 -- Advantech PCL-818 cards, PCL-718

Author: Michal Dobes <dobes@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)
```

5.6.91 pcm3724 -- Advantech PCM-3724

Author: Drew Csillag <drew_csillag@yahoo.com>

Status: tested

Manufacturer	Device	Name
Advantech	PCM-3724	pcm724

```
This is driver for digital I/O boards PCM-3724 with 48 DIO.
It needs 8255.o for operations and only immediate mode is supported.
See the source for configuration details.

Copy/pasted/hacked from pcm724.c
```

5.6.92 pcm3730 -- PCM3730

Author: Blaine Lee

Status: unknown

Manufacturer	Device	Name
Advantech	PCM-3730	pcm3730

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

5.6.93 pcmad -- 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
```

5.6.94 pcmda12 -- A driver for the Winsystems PCM-D/A-12

Author: Calin Culianu <calin@ajvar.org>

Status: works

Manufacturer	Device	Name
Winsystems	PCM-D/A-12	pcmda12

```
A driver for the relatively straightforward-to-program PCM-D/A-12.
This board doesn't support commands, and the only way to set its
analog output range is to jumper the board.  As such,
comedi_data_write() ignores the range value specified.

The board uses 16 consecutive I/O addresses starting at the I/O port
base address.  Each address corresponds to the LSB then MSB of a
particular channel from 0-7.

Note that the board is not ISA-PNP capable and thus
needs the I/O port comedi_config parameter.

Note that passing a nonzero value as the second config option will
enable "simultaneous xfer" mode for this board, in which AO writes
will not take effect until a subsequent read of any AO channel.  This
is so that one can speed up programming by preloading all AO registers
with values before simultaneously setting them to take effect with one
read command.

Configuration Options:
  [0] - I/O port base address
  [1] - Do Simultaneous Xfer (see description)
```

5.6.95 pcmmio -- A driver for the PCM-MIO multifunction board

Author: Calin Culianu <calin@ajvar.org>

Status: works

Manufacturer	Device	Name
Winsystems	PCM-MIO	pcmmio

```
A driver for the relatively new PCM-MIO multifunction board from
Winsystems.  This board is a PC-104 based I/O board.  It contains
four subdevices:
  subdevice 0 - 16 channels of 16-bit AI
```

```
  subdevice 1 - 8 channels of 16-bit AO
  subdevice 2 - first 24 channels of the 48 channel of DIO (with edge-triggered interrupt ↩
      support)
  subdevice 3 - last 24 channels of the 48 channel DIO (no interrupt support for this bank ↩
      of channels)

  Some notes:

  Synchronous reads and writes are the only things implemented for AI and AO,
  even though the hardware itself can do streaming acquisition, etc.  Anyone
  want to add asynchronous I/O for AI/AO as a feature?  Be my guest...

  Asynchronous I/O for the DIO subdevices *is* implemented, however!  They are
  basically edge-triggered interrupts for any configuration of the first
  24 DIO-lines.

  Also note that this interrupt support is untested.

  A few words about edge-detection IRQ support (commands on DIO):

  * To use edge-detection IRQ support for the DIO subdevice, pass the IRQ
    of the board to the comedi_config command.  The board IRQ is not jumpered
    but rather configured through software, so any IRQ from 1-15 is OK.

  * Due to the genericity of the comedi API, you need to create a special
    comedi_command in order to use edge-triggered interrupts for DIO.

  * Use comedi_commands with TRIG_NOW.  Your callback will be called each
    time an edge is detected on the specified DIO line(s), and the data
    values will be two sample_t's, which should be concatenated to form
    one 32-bit unsigned int.  This value is the mask of channels that had
    edges detected from your channel list.  Note that the bits positions
    in the mask correspond to positions in your chanlist when you
    specified the command and *not* channel id's!

 *  To set the polarity of the edge-detection interrupts pass a nonzero value
    for either CR_RANGE or CR_AREF for edge-up polarity, or a zero
    value for both CR_RANGE and CR_AREF if you want edge-down polarity.

Configuration Options:
  [0] - I/O port base address
  [1] - IRQ (optional -- for edge-detect interrupt support only, leave out if you don't ↩
      need this feature)
```

5.6.96 pcmuio -- A driver for the PCM-UIO48A and PCM-UIO96A boards from Winsystems.

Author: Calin Culianu <calin@ajvar.org>

Status: works

Manufacturer	Device	Name
Winsystems	PCM-UIO48A	pcmuio48
Winsystems	PCM-UIO96A	pcmuio96

```
A driver for the relatively straightforward-to-program PCM-UIO48A and
PCM-UIO96A boards from Winsystems.  These boards use either one or two
(in the 96-DIO version) WS16C48 ASIC HighDensity I/O Chips (HDIO).
This chip is interesting in that each I/O line is individually
programmable for INPUT or OUTPUT (thus comedi_dio_config can be done
on a per-channel basis).  Also, each chip supports edge-triggered
interrupts for the first 24 I/O lines.  Of course, since the
96-channel version of the board has two ASICs, it can detect polarity
changes on up to 48 I/O lines.  Since this is essentially an (non-PnP)
ISA board, I/O Address and IRQ selection are done through jumpers on
the board.  You need to pass that information to this driver as the
first and second comedi_config option, respectively.  Note that the
48-channel version uses 16 bytes of IO memory and the 96-channel
version uses 32-bytes (in case you are worried about conflicts).  The
48-channel board is split into two 24-channel comedi subdevices.
The 96-channel board is split into 4 24-channel DIO subdevices.

Note that IRQ support has been added, but it is untested.

To use edge-detection IRQ support, pass the IRQs of both ASICS
(for the 96 channel version) or just 1 ASIC (for 48-channel version).
Then, use use comedi_commands with TRIG_NOW.
Your callback will be called each time an edge is triggered, and the data
values will be two sample_t's, which should be concatenated to form one
32-bit unsigned int.  This value is the mask of channels that had
edges detected from your channel list.  Note that the bits positions
in the mask correspond to positions in your chanlist when you specified
the command and *not* channel id's!

To set the polarity of the edge-detection interrupts pass a nonzero value for
either CR_RANGE or CR_AREF for edge-up polarity, or a zero value for both
CR_RANGE and CR_AREF if you want edge-down polarity.

In the 48-channel version:

On subdev 0, the first 24 channels channels are edge-detect channels.

In the 96-channel board you have the collowing channels that can do edge detection:

subdev 0, channels 0-24  (first 24 channels of 1st ASIC)
subdev 2, channels 0-24  (first 24 channels of 2nd ASIC)

Configuration Options:
  [0] - I/O port base address
  [1] - IRQ (for first ASIC, or first 24 channels)
  [2] - IRQ for second ASIC (pcmuio96 only - IRQ for chans 48-72 .. can be the same as  ↩
      first irq!)
```

5.6.97 poc -- Generic driver for very simple devices

Author: ds

Status: unknown

Manufacturer	Device	Name
Keithley Metrabyte	DAC-02	dac02

Manufacturer	Device	Name
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
```

5.6.98 quatech_daqp_cs -- Quatech DAQP PCMCIA data capture cards

Author: Brent Baccala <baccala@freesoft.org>

Status: works

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

5.6.99 rtd520 -- 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	rtd520
Real Time Devices	DM7520HR-8	rtd520
Real Time Devices	PCI4520	rtd520
Real Time Devices	PCI4520-8	rtd520

```
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)
```

5.6.100 rti800 -- 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]
  [6] - DAC 0 encoding
          0 = two's complement
          1 = straight binary
  [7] - DAC 1 range (same as DAC 0)
  [8] - DAC 1 encoding (same as DAC 0)
```

5.6.101 rti802 -- 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 ...
```

5.6.102 s526 -- Sensoray 526 driver

Author: Richie Everett Wang <everett.wang@everteq.com>

Status: experimental

Manufacturer	Device	Name
Sensoray	526	s526

```
Encoder works
Analog input works
Analog output works
PWM output works
Commands are not supported yet.

Configuration Options:

comedi_config /dev/comedi0 s526 0x2C0,0x3
```

5.6.103 s626 -- Sensoray 626 driver

Author: Richie Everett Wang <everett.wang@everteq.com>

Status: experimental

Manufacturer	Device	Name
Sensoray	626	s626

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

INSN_CONFIG instructions:
  analog input:
   none

  analog output:
   none

  digital channel:
   s626 has 3 dio subdevices (2,3 and 4) each with 16 i/o channels
   supported configuration options:
   INSN_CONFIG_DIO_QUERY
   COMEDI_INPUT
```

```
   COMEDI_OUTPUT

  encoder:
   Every channel must be configured before reading.

   Example code

   insn.insn=INSN_CONFIG;   //configuration instruction
   insn.n=1;                //number of operation (must be 1)
   insn.data=&initialvalue; //initial value loaded into encoder
                            //during configuration
   insn.subdev=5;           //encoder subdevice
   insn.chanspec=CR_PACK(encoder_channel,0,AREF_OTHER); //encoder_channel
                                                        //to configure

   comedi_do_insn(cf,&insn); //executing configuration
```

5.6.104 serial2002 -- Driver for serial connected hardware

Author: Anders Blomdell

Status: in development

5.6.105 skel -- 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
```

5.6.106 ssv_dnp -- 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

5.6.107 unioxx5 -- Driver for Fastwel UNIOxx-5 (analog and digital i/o) boards.

Author: Kruchinin Daniil (asgard) <asgard@etersoft.ru>

Status: unknown

Manufacturer	Device	Name
Fastwel	UNIOxx-5	unioxx5
Fastwel	UNIOxx-5	unioxx5

```
This card supports digital and analog I/O. It written for g01
subdevices only.
channels range: 0 .. 23 dio channels
and 0 .. 11 analog modules range
During attaching unioxx5 module displays modules identifiers
(see dmesg after comedi_config) in format:
| [module_number] module_id |
```

5.6.108 usbdux -- Driver for USB-DUX-D of INCITE Technology Limited

Author: Bernd Porr <tech@linux-usb-daq.co.uk>

Status: Stable

Manufacturer	Device	Name
ITL	USB-DUX-D	usbdux

```
The following subdevices are available
  - Analog input
    subdevice: 0
    number of channels: 8
    max data value: 4095
    ranges:
      all channels:
        range = 0 : [-4.096 V,4.096 V]
        range = 1 : [-2.048 V,2.048 V]
        range = 2 : [0 V,4.096 V]
        range = 3 : [0 V,2.048 V]
    command:
      start: now|int
      scan_begin: timer (contains the sampling interval. min is 125us / chan)
      convert: now
      scan_end: count
      stop: none|count
```

```
  - Analogue output:
    subdevice: 1
    number of channels: 4
    max data value: 4095
    ranges:
      all channels:
        range = 0 : [-4.096 V,4.096 V]
        range = 1 : [0 V,4.096 V]
    command:
      start: now|int
      scan_begin: timer (contains the sampling interval. min is 1ms.)
      convert: now
      scan_end: count
      stop: none|count
  - Digital I/O
    subdevice: 2
    number of channels: 8
  - Counter
    subdevice: 3
    number of channels: 4
    max data value: 65535
    Pin assignments on the D-connector:
      0=/CLK0, 1=UP/DOWN0, 2=RESET0, 4=/CLK1, 5=UP/DOWN1, 6=RESET1
  - PWM
    subdevice: 4
    number of channels: 8 or 4 + polarity output for H-bridge
                             (see INSN_CONFIG_PWM_SET_H_BRIDGE where
                              the first byte is the value and the
                              second the polarity)
    max data value: 512

Configuration options
  The device requires firmware which is usually
  uploaded automatically by udev/hotplug at the moment
  the driver module is loaded.
  In case udev/hotplug is not enabled you need to upload
  the firmware with comedi_config -i usbdux_firmware.bin.
  The firmware is usually installed under /lib/firmware
  or can be downloaded form http://www.linux-usb-daq.co.uk.
```

5.6.109 usbduxfast -- Driver for USB-DUX-FAST of INCITE Technology Limited

Author: Bernd Porr <tech@linux-usb-daq.co.uk>

Status: stable

Manufacturer	Device	Name
ITL	USB-DUX-FAST	usbduxfast

```
The device has one subdevice for analogue input.
  - subdevice: 0
    number of channels: 16
    max data value: 4096
    ranges:
```

```
      all channelss:
        range = 0 : [-0.75 V,0.75 V]
        range = 1 : [-0.5 V,0.5 V]
    command:
      The channel-list allows 1,2,3 and 16 channels.
      start: now|ext|int (external trigger via pin at HD-D connector)
      scan_begin: follow|timer|ext
      convert: timer|ext (contains the sampling interval. Min interval
                          for single channel acquisition is 33us
                          and for multiplexed acquisition 300us)
      scan_end: count
      stop: none|count

Configuration options:
  The device requires firmware which is usually
  uploaded automatically by udev/hotplug at the moment
  the driver module is being loaded.
  In case udev/hotplug is not enabled you need to upload
  the firmware with comedi_config -i usbduxfast_firmware.bin
  The firmware is usually installed under /lib/firmware
  or can be downloaded form http://www.linux-usb-daq.co.uk.
```

5.6.110 usbduxsigma -- Driver for USB-DUX-SIGMA of INCITE Technology Limited

Author: Bernd Porr <tech@linux-usb-daq.co.uk>

Status: Stable

Manufacturer	Device	Name
ITL	USB-DUX-SIGMA	usbduxsigma

```
The following subdevices are available
  - Analog input
    subdevice: 0
    number of channels: 16
    max data value: 16777215 (0xffffff, 24bits)
    ranges:
      all channels: [-1.325 V,1.325 V]
    command:
      start: now|int
      scan_begin: timer (contains the sampling interval. min 250us)
      convert: now
      scan_end: count
      stop: none|count
  - Analog output
    subdevice: 1
    number of channels: 4
    max data value: 255
    ranges:
      all channels: [0 V,2.5 V]
    command:
      start: now|int
      scan_begin: timer (contains the sampling interval. min 1ms)
      convert: now
```

```
      scan_end: count
      stop: none|count
  - Digital I/O
    subdevice: 2
    number of channels: 24 (first 8 bits on the D connector, 16 bits int.)
  - PWM
    subdevice: 3
    number of channels: 8 or 4 + polarity output for H-bridge
                            (see INSN_CONFIG_PWM_SET_H_BRIDGE where
                             the first byte is the value and the
                             second the polarity)
    max data value: 512

Configuration options:
  The device requires firmware which is usually
  uploaded automatically by udev/hotplug at the moment
  the driver module is loaded.
  In case udev/hotplug is not enabled you need to upload
  the firmware with comedi_config -i usbdux_firmware.bin.
  The firmware is usually installed under /lib/firmware
  or can be downloaded form http://www.linux-usb-daq.co.uk.
```

6 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.

6.1 Communication user-space — kernel-space

In user-space, you interact with the functions implemented in the Comedilib library. 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 `lib/ioctl.c` in Comedilib.) 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 `comedi_fops.c` file in the Comedi sources: the `comedi_unlocked_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()` function; acquisition data captured by the interface card passes the kernel/user-space boundary with the help of a `copy_to_user()` function.

6.2 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 (`kcomedilib` module).

From all the relevant Comedi device driver code that is found in the `comedi` kernel module source directory, 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.

6.2.1 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 `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: range (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).

6.2.1.1 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];
};
```

6.2.1.2 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,  ↩
      unsigned int start_chan_index );

  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* function pointers 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*.

6.2.1.3 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);
};
```

6.2.1.4 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);
};
```

6.2.1.5 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;
};
```

6.2.2 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)(unsigned int,void *),void *arg);

  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 long flags,const char *device,comedi_device *dev_id);
void comedi_free_irq(unsigned int irq,comedi_device *dev_id)
```

6.3 Board-specific functionality

The 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 (negative) 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 mydriver_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 include/linux/ directory.

Drivers with digital I/O subdevices should implement the following functions, setting the function pointers in the comedi_subdevice:

- 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 `insn_read`() and `insn_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...

6.4 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" (or "End-Of-Output").
- `COMEDI_CB_EOS`: execute the callback at the "End-Of-Scan".
- `COMEDI_CB_OVERFLOW`: execute the callback when a buffer overflow or underflow has occurred.
- `COMEDI_CB_ERROR`: execute the callback at the occurrence of an (undetermined) error.

6.5 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 (apart from read-only, constant data, or data structures shared by all devices), 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.)

6.6 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 sensible 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/drivers/Makefile.am` and add `mydriver.ko` to the `module_PROGRAMS` list. Also add a line

```
mydriver_ko_SOURCES = mydriver.c
```

  in the alphabetically appropriate place.
- Edit `comedi/drivers/Kbuild` and a line according to the type of driver:

```
obj-$(COMEDI_CONFIG_PCI_MODULES) += mydriver.o  # for a PCI driver
```

```
obj-$(COMEDI_CONFIG_PCMCIA_MODULES) += mydriver.o  # for a PCMCIA driver
```

```
obj-$(COMEDI_CONFIG_USB_MODULES) += mydriver.o  # for a USB driver
```

```
obj-m += mydriver.o  # for other driver types
```

- Run **./autogen.sh** in the top-level comedi directory. You will need to have (a recent version of) autoconf and automake installed to successfully run **autogen.sh**. Afterwards, your driver will be built along with the rest of the drivers when you run **make**.
- If you want to have your driver included in the Comedi distribution (you *definitely* want to :-)) send it to the Comedi mailing list for review and integration. See the top-level `README` for details of the Comedi mailing list.) Note your work must be licensed under terms compatible with the GNU GPL to be distributed as a part of Comedi.

7 Glossary

Application Program Interface (`API`)

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 functions `comedi_set_buffer_size`() and `comedi_set_max_buffer_size`() 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_set_buffer_size()` for more information.

Differential IO

...

Direct Memory Access (`DMA`)

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 (`FIFO`)

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 the **comedi_config** utility 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.