COMEDI and UIO Drivers for PCI Multifunction Data Acquisition and Generic I/O Cards and Their QEMU Virtual Hardware Equivalents

Pavel Píša
Czech Technical University in Prague, Department of Control Engineering
Karlovo náměstí 13, 121 35 Praha 2, Czech Republic
pisa@cmp.felk.cvut.cz

Rostislav Lisový
Czech Technical University in Prague, Faculty of Electrical Engineering
Karlovo náměstí 13, 121 35 Praha 2, Czech Republic
lisovy@gmail.com

Abstract

The article describes implementation of UIO and Comedi drivers for Humusoft MF624 and MF614 data acquisition cards. Basic functions (D/A, A/D converters, digital inputs/outputs) of Humusoft MF624 card were implemented into the Qemu emulator as well which enable to experiment with drivers implementation without physical access to the cards and risk of data lost when drivers are developed and tested on same primary Linux kernel instance. The article can help newcomers in the area to gain knowledge required to implement support for other similar cards and hardware emulation of these cards. The matching real and virtual setup can be used in operating system courses for practical introduction to simple drivers implementation and helps with understanding internal computation world with real world computers interfacing.

1 Introduction

When teaching development of Linux drivers one of the approaches is to explain kernel API and programming paradigms by creating driver which does not require any special hardware – e.g. character driver which returns upper case ASCII text when receiving lower case. Although this approach can be useful, the issues associated with dealing with hardware should be practised as well.

The approach we took in this work eliminates the need of physical access to hardware whereas it provides full feature set of PCI device in form of virtual hardware. This was possible by implementing virtual PCI device into Qemu emulator.

The main reason of choosing DAQ cards for this project was easy interfacing from programmer's point of view and straightforward testing of proper function of the driver. This can be very helpful for beginners who are not familiar with hardware related topics.

2 Humusoft MF614, MF624

Humusoft MF614 and MF624 are data acquisition (DAQ) cards. Both of these cards use PCI interface to connect to the computer. The main features this cards provide are *digital inputs*, *digital outputs*, *ADCs*, *DACs*, *timers*, *encoder inputs*. Humusoft MF614 is predecessor of MF624 – available functions are quite similar. The main difference is in driver programming – MF614 has only 8-bit wide registers, whereas MF624 has ones 16- or 32-bit wide.

MF624 is available for purchase on manufacturer's web page. MF614 is no more produced.

3 UIO Driver

Each UIO driver consists of two parts – small kernel module (the need for it is mostly because of device-specific interrupt handling/disabling) and user-space driver logic (as shown in figure 1). The main advantage of this approach is that the most of the development happens in user-space, thus during prototyping the driver (or when using a bad one) the integrity and stability of the kernel will not be disrupted.

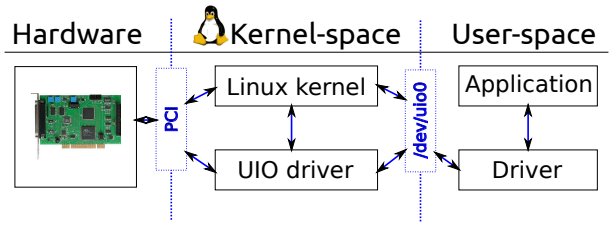


FIGURE 1: *UIO driver structure*

Driver `uio_pci_generic`

When dealing with any device compliant to PCI 2.3, it is also possible to use `uio_pci_generic` driver in kernel instead of programming a specific one. This driver makes all memory regions of the device available to user-space.

Binding to the device is done by writing Vendor and Device ID into `/sys/bus/pci/drivers/uio_pci_generic/new_id` file.

Interrupt handler uses Interrupt Disable bit in the PCI command register and Interrupt Status bit in the PCI status register. Because neither of MF614 or MF624 is PCI 2.3 compliant it is not possible to use this driver for them.

Implementing the Kernel Part

In case of writing UIO driver for PCI device, initialization function of the module registers `struct pci_driver` in standard way[1], where the probe function handles initialization of UIO-related structures. The main structure holding all data of particular UIO driver is `struct uio_info`. Its simple initialization (including registration) is shown below:

```
 1 | /* struct pci_dev *dev */
 2 | struct uio_info *info;
 3 | info = kzalloc(sizeof(struct uio_info),
 4 |   GFP_KERNEL);
 5 |
 6 | info->name = "mf624";
 7 | info->version = "0.0.1";
 8 |
 9 | info->mem[0].name = "PCI chipset";
10 | info->mem[0].addr =
11 |   pci_resource_start(dev, 0);
12 | info->mem[0].size =
13 |   pci_resource_len(dev, 0);
14 | info->mem[0].memtype = UIO_MEM_PHYS;
15 | info->mem[0].internal_addr =
16 |   pci_ioremap_bar(dev, 0);
17 |
18 | info->port[0].name =
19 |   "Board programming registers";
20 | info->port[0].porttype = UIO_PORT_X86;
21 | info->port[0].start =
22 |   pci_resource_start(dev, 1);
23 | info->port[0].size =
24 |   pci_resource_len(dev, 1);
25 |
26 | uio_register_device(&dev->dev, info);
27 | pci_set_drvdata(dev, info);
```

Structure `uio_mem` is used for enabling memory-mapped I/O regions, whereas structure `uio_port` is used for I/O ports (for each of these structures there is statically allocated array with a size of 5 elements).

Interface to User-space

Communication with kernel part of the UIO driver is possible through `/dev/uioX` file (where X is the number of instance of a driver). There are several syscalls possible to be used when interfacing with this file:

`open()` opens the device, returns file descriptor used for another syscalls.

`read()` blocks until an interrupt occurs (the value read is number of interrupts seen by the device).

`mmap()` is used to map memory of the device to user-space. The offset value passed to `mmap()` determines the memory area of a device to map – for *n-th* area offset should be *n*sysconf(_SC_PAGESIZE)*.

[1] For more information about PCI driver development see [1] available online at *https://lwn.net/Kernel/LDD3/*

`irqcontrol()` is used for enabling (called with parameter set to `(int) 1`) or disabling (`(int) 0`) interrupts.

It is possible to define your own `mmap()`, `open()`, `release()` functions as an option. When there is need to use `irqcontrol()`, it is necessary to implement this function per device.

Information related to a particular driver instance can be found in `/sys/class/uio/uioX` directory. Most of the files are read-only. The subdirectory `maps` contains information about MMIO regions mapped by the driver, subdirectory `portio` is for I/O port regions.

When using UIO and `mmap()` with MF624 card (which has 32 or 128 bytes long memory regions) there is an issue with the return value of this syscall – the pointer to the memory is page-size-aligned, so it is necessary to add low bits of physical address (page offset) of each memory region to it. Physical address can be obtained from `addr` file located in `/sys/class/uio/uioX/maps/mapX`. Region offset is equal to `addr & (sysconf(_SC_PAGESIZE) - 1)`.

4 Comedi Driver

UIO driver is a versatile solution available mainly for uncommon devices. In our case of using DAQ card a special subsystem in Linux kernel designated for DAQ card drivers can be used. It is called Comedi (*Linux control and measurement device interface*). It provides library functions for user- and kernel-space making development and usage of DAQ devices easier. It consists of three different parts.

Comedi is a part of Linux kernel. It consist of individual device drivers including Comedi driver providing basic set of functions used by device drivers.

Comedilib is a user-space library providing unified interface for another user-space application to devices supported by Comedi.

Kcomedilib is also a part of Linux kernel. It provides the same API as Comedilib, whereas this is used for real-time applications.

Implementing the Driver

Each Comedi driver should be registered to the list of active Comedi drivers. This is done by invoking `comedi_driver_register()` function. The only parameter passed to this function is pointer to `struct comedi_driver` structure. The most important fields of this structure are:

```
const char *driver_name; /* "my_driver" */
struct module *module;   /* THIS_MODULE */
int (*attach) (struct comedi_device *,
  struct comedi_devconfig *);
int (*detach) (struct comedi_device *);
```

Unlike the UIO or generic PCI driver, the main *initialization function* is not `probe()` (of `struct pci_driver`) but `attach()` (of `struct comedi_driver`) which is invoked by Comedi subsystem.

The `attach()` function is responsible not only for common PCI device initialization but also for initialization of `struct comedi_device` (which is accessible through a pointer passed to `attach()` function). The most important step is to allocate and initialize each *subdevice* (in Comedis nomenclature *subdevice* represents one particular function of the device – e.g. ADC, digital out, etc.) of the DAQ card. Allocation is done by Comedi function `alloc_subdevices(struct comedi_device *dev, unsigned int num_subdev)`, each `struct comedi_subdevice` is then accessible in array called `subdevices` which is part of `struct comedi_device`. Example of initialization of subdevice representing ADC:

```
 1 | s = dev->subdevices + 0;
 2 | s->type = COMEDI_SUBD_AI;
 3 | s->subdev_flags = SDF_READABLE |
 4 |                   SDF_GROUND;
 5 | s->n_chan = 8;
 6 | s->maxdata = (1 << 14) - 1;
 7 | s->range_table = &range_bipolar10;
 8 | s->len_chanlist = 8;
 9 | s->insn_read = mf624_ai_rinsn;
10 | s->insn_config = mf624_ai_cfg;
```

Interface to User-space

After successful compilation and loading of particular Comedi driver, there should be `/dev/comediX` (where X is number of instance of a driver) file. For communication with this file Comedi library functions are used. For opening device – `comedi_open()`, for reading/writing ADCs/DACs – `comedi_data_read()`, `comedi_data_write()` and for reading/writing digital inputs/outputs – `comedi_dio_read()`, `comedi_dio_write()`.

There are already applications using Comedi API[2] – thus in some cases there is no need for implementing user-space application from scratch.

5 Qemu Virtual Hardware

Qemu is an open-source processor emulator. Unlike common virtualization solutions it is able of emulating x86, x86-64, ARM and other widespread processor architectures. For the purposes of this work it was used for implementing virtual Humusoft MF624 DAQ card.

Implementation of Virtual PCI device

When creating new virtual device in Qemu, main hook into Qemu device infrastructure is done by invoking `device_init()` with parameter of pointer to initialization function with prototype of `static void (*)(void)`. For registering new PCI device, it is necessary to call `pci_qdev_register()` passing parameter of pointer to `PCIDeviceInfo`. The most important fields of this Qemu-specific data type are pointers to *init* and *exit* functions with prototype of `int (*)(PCIDevice *)`.

The PCI device specific initialization consists of:

- Initializing configuration space of PCI device – e.g. setting Vendor and Device IDs, device class, interrupt pin, etc.
- Registration of I/O memory used by the device.
- Creating a function (called when device gets allocated memory from virtual PCI controller) for mapping of physical memory to particular *BARs* (Base Address Registers) of the PCI device.

The very basic (non-compilable) example:

```
 1 | static CPUReadMemoryFunc * const mf624_BAR0_read[3] = { NULL, NULL, mf624_BAR0_read32 };
 2 | static CPUWriteMemoryFunc * const mf624_BAR0_write[3] = { NULL, NULL, mf624_BAR0_write32 };
 3 |
 4 | static void mf624_map(PCIDevice *pci_dev, int region, pcibus_t addr, pcibus_t sz, int tp)
 5 | {
 6 |   mf624_state_t *s = DO_UPCAST(mf624_state_t, dev, pci_dev);
 7 |   cpu_register_physical_memory(addr + 0x0, BAR0_size, s->BAR0_mem_table_index);
 8 | }
 9 |
10 | static int pci_mf624_init(PCIDevice *pci_dev)
11 | {
12 |   mf624_state_t *s = DO_UPCAST(mf624_state_t, dev, pci_dev); /* i.e. container_of() */
13 |   uint8_t *pci_conf;
14 |
15 |   pci_conf = s->dev.config;
16 |   pci_config_set_vendor_id(pci_conf, PCI_VENDOR_ID_HUMUSOFT);
17 |   /* ... */
18 |   pci_conf[PCI_INTERRUPT_PIN] = 0x1;
19 |
20 |   s->BAR0_mem_table_index = cpu_register_io_memory(mf624_BAR0_read, mf624_BAR0_write,
21 |                             s, DEVICE_NATIVE_ENDIAN); /* returns unsigned int */
22 |   pci_register_bar(&s->dev, 0, BAR0_size, PCI_BASE_ADDRESS_SPACE_MEMORY, mf624_map);
23 |   return 0;
24 | }
25 |
26 | static PCIDeviceInfo mf624_info = {
27 |   .qdev.name = "mf624", .qdev.size = sizeof(mf624_state_t),
28 |   .init = mf624_init,   .exit = mf624_exit,
29 | };
30 | static void reg_dev(void) { pci_qdev_register(&mf624_info); }
31 | device_init(reg_dev)
```

[2]For basic list of available applications see http://www.comedi.org/applications.html

Usage of Virtual MF624

When running any guest operating system in Qemu (with support for MF624 activated) the virtual MF624 device is available in the same way as if it was real hardware – there are no issues with interfacing between guest operating system and virtual device. Interfacing between virtual hardware and *real world* is handled by TCP/IP connection from MF624 part in Qemu to *host* operating system. It is used for reading/setting output/input values (as shown in figure 2).

The most fundamental way of communication through this channel is by using `telnet` application. Example of real communication:

```
$ telnet localhost 55555
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
DA1=9.998779
DOUT=255.000000
DOUT=0.000000
DA1=5.000000
^]
telnet> Connection closed.
```

As a much more easier way of interfacing, there is also graphical application created just for purposes of communication with virtual MF624 card (see figure 3). It was created using Qt4 graphical toolkit.

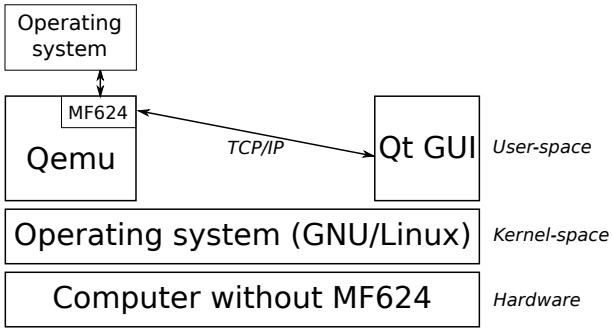


FIGURE 2: *Qemu implementing virtual MF624 device*

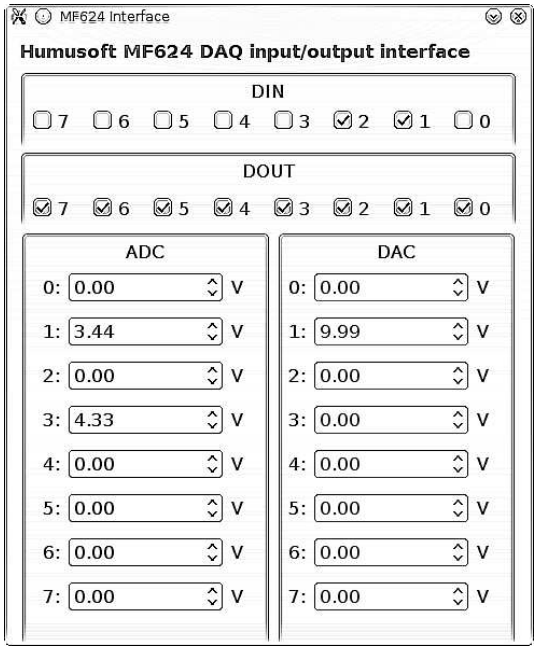


FIGURE 3: *Graphical application used for interfacing between virtual MF624 and real world*

6 Conclusion

The outcome of this work creates basic integrated tool for teaching PCI driver development (mostly) for GNU/Linux operating system. Its main advantage is the possibility to train driver development on *real hardware* without the necessity of having an expensive DAQ device. The other advantage is a safe environment for driver prototyping – where no mistake can damage host operating system.

All the information (including source code) related to topic covered in this article are publicly available on web page `rtime.felk.cvut.cz/hw/index.php/Humusoft_MF6xx`

References

[1] *Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, Linux Device Drivers, 3rd Edition, O'Reilly Media, 2005*

[2] *Hans-Jürgen Koch http://www.kernel.org/doc/htmldocs/uio-howto.html*

[3] *David Schleef, Frank Hess, Herman Bruyninckx http://www.comedi.org/doc/*

[4] *Fabrice Bellard et al. git://git.qemu.org/qemu.git*