CMIDID Kernel Driver
Module Setup and GPIO Input

Praktikum Linux & C - Technische Universität München

Michael Opitz

August 3, 2014

1 Introduction

In this part of the documentation, the setup of the CMIDID driver and the distribution into several components will be explained, as well as the implementation of the GPIO component, which is the one that handles the input on GPIO ports and translates that input into MIDI events. I spent most of my time during the development phase working on the latter component, but before we actually started to implement the driver, I looked into the high-resolution timer API of the kernel. That was because we knew from the beginning that measuring the time difference between consecutive key hits (of our breadboard MIDI keyboard) would be required. Fortunately, the hrtimer API turned out to be very user-friendly and in the end we only required some simple functions like

```
1 ktime_get();
2 hrtimer_start(struct hrtimer *
    timer, ktime_t tim, const
    enum hrtimer_mode mode);
3 hrtimer_init(struct hrtimer *
    timer, clockid_t which_clock,
     enum hrtimer_mode mode);
```

and a couple more (see: https://github.com/torvalds/linux/blob/master/include/linux/hrtimer.h). So, we decided that we don't need a separate module component to handle the timer related functionality and included this in the GPIO component.

So, besides the timer component, which was not required, and the GPIO component I already mentioned, we did add a main component, which handles initialization of the other components, as well as a MIDI component, which is responsible for creating a virtual MIDI device, and for outputting MIDI events on that device.

In the section **Project Setup**, I'm going to show how the compilation of the several module components works and how the component initialization and exit routines have to be handled. The other part of his documentation is a detailed explanation of the GPIO component: **Working with GPIOs**. This includes the steps necessary to read input on the GPIO ports from the kernel module, a description of a clean exit routine, as well as a couple of special cases and workaround for bugs, that are worth mentioning. The GPIO component has a significant amount of code for handling

module customization via `ioctl` and module parameters; this won't be explained in this part of the documentation, but rather in (TODO: CITE JANNIK). And for an in-depth guide on the MIDI component, take a look at (TODO: CITE ANDI).

2 Project Setup

The CMIDID kernel driver found in the `module` subdirectory, is split up into the following components. The following enumeration and the following subsections contain a short overview for each of those components as well as short explanation of our decision as to why we needed/created those components.

- The main component: `cmidid_main.c`. That's the one that handles the actual module initialization and the module cleanup, aw well as the creation of a character device for `ioctl`, and the `ioctl` callback function. See section **The Main Module Component**.
- The MIDI component: `cmidid_midi.c`. This component handles the creation of the virtual MIDI soundcard via the ALSA sequencer interface, and the sending of MIDI events (in our case only the events `noteon` and `noteoff`). This is component is explained in Andreas' part of the documentation.
- The GPIO component: `cmidid_gpio.c`. This component is responsible for reading input on the GPIO ports, for translating the input into MIDI keyboard key hit events and for informing the MIDI component to actually send a MIDI event. See section **The GPIO Component**.

In-tree modules are usually contained in a signle `.c` file, but we used this setup to make a clean seperation of the individual logical parts of the module and to increase readability of each single source file. We added a header for every single component, where the `cmidid_ioctl.h` header is special, in the way that it needs to be included by userspace programs that want to communicate with the module via `ioctl`. And the `cmidid_util.h` header contains only a couple of `printk` macros to ease the output of debug information into the kernel log. The final structure of our project setup looks like this:

```
module/
  Makefile
  cmidid_main.h
  cmiddi_main.c
  cmidid_gpio.h
  cmidid_gpio.c
  cmidid_midi.h
  cmidid_midi.c
  cmidid_ioctl.h
  cmidid_util.h
```

Additional to following overviews of each component, the next section provides a brieft introduciton of the KBuild system and how we used this to compile the components into a single module.

2.1 Compilation with Kbuild

The Linux kernel uses a system named Kbuild which is thoroughly described in https://github.com/torvalds/linux/blob/master/Documentation/kbuild/. For building external (i.e. out-of-tree) modules, the documentation file https://github.com/torvalds/linux/blob/master/Documentation/kbuild/modules.txt was especially relevant and

explained the creation of makefiles for this type of module.

For compiling a single out-of-tree module that consists of several components, one can provide the name of the module and the names of the components with the following syntax to Kbuild:

```
obj-m := cmidid.o
cmidid-objs := cmidid_midi.o
    cmidid_main.o cmidid_gpio.o
```

Kbuild will derive the required source files from the given information. The only thing left to do is to build the actual module with

```
all:
  $(MAKE) -C $(KSRC) M=$$PWD
    modules
```

The `-C` flag tells `make` to change in the directory containing the sources for the running kernel, and the `M` option is required for the location of the external module that we are compiling.

2.2 The Main Module Component

The component `cmidid_main.c` of the CMIDID module does the following:

- Setting up the required module information, like the init and exit function with the `module_init` and `module_exit` macros.
- Creation of a character device in the initialization routine. This device is required for `ioctl` and it's created with the `cdev` interface. First the character device needs to be created with the subsequent calls `alloc_chrdev_region` (gets the device number), `cdev_alloc` (allocates space for the actual device) and `cdev_add` (adds the device to the system). Finally, for the character device to appear in the sysfs filesystem, we need can create the device node from our module with `device_create`.
- Handling of the initialization routine for every other module component. There's one problem that needs to be taken care of: If the GPIO component is initialized after the MIDI component, and if the initialization of the MIDI component fails, we can't just call `exit(-1)` to terminate the module. Every memory allocate in the MIDI init routine needs to be freed manually, so that we can exit without memory leaks. The most readable way to do this, is to call the cleanup functions in inverse order via a list of `goto` labels like this:

```
if ((err = cmidid_midi_init
    () < 0)
  goto err_midi_init;

if ((err = cmidid_gpio_init(
    ) < 0)
  gotot err_gpio_init;

err_gpio_init:
  cmidid_midi_exit();

err_midi_init:
  free character device
  ...
```

- The `exit` routine of the module needs to call the subcomponent cleanup routines in a similar fashion to guarantee a clean exit.
- Additional to the above, the main component also handles the `ioctl` user in-

put. This process is explained more detailed in Jannik's part of the documentation.

2.3 The GPIO Component

3 Working with GPIOs

In this section, the GPIO component of the CMIDID driver is described in more detail. The component consists for the first part of the header `cmidid_gpio.h` which provides several function definitions for the main module, including the init and exit routines of the GPIO component, and functions for `ioctl` configuration. The corresponding source file is `cmidid_gpio.c`. The following subsections describe some of the more interesing parts of the component. A definition of the GPIO interface of the kernel can be found in the header file https://github.com/torvalds/linux/blob/master/include/linux/gpio.h.

3.1 The `key` Struct

The main abstraction that is used in the GPIO component, is the `struct key` which describes an abstract key of a MIDI keyboard. Here's the definition:

```
1  struct key {
2    KEY_STATE state;
3    struct gpio gpios[2];
4    unsigned int irqs[2];
5    ktime_t hit_time;
6    bool timer_started;
7    struct hrtimer hrtimers[2];
8    unsigned char note;
9    int last_velocity;
10 };
```

Each key of our keyboard has two buttons which are required to determine the velocity/force of the key hit, so we need a pair of GPIO ports for every button. Most of the other struct members are necessary to actually calculate the hit velocity for the specific key and the `note` member associates a MIDI note with the button.

3.2 GPIO Interrupts

Fortunately, the Raspberry Pi supported interrupts for every GPIO pin, so it was not necessary to write extra code to listen on each port for rising and falling edge triggers. We could rather specify a callback function for every IRQ: `irq_handler` in `cmidid_gpio.c`. That's the function we used to calculate the actual time difference between the button hits for each keyboard key.

3.3 Initialization

The GPIO initialization function `cmidid_gpio_init` takes care of of several required steps to setup working GPIO ports.

- The number of required GPIOs is determined via the `gpio_mapping` module parameter.
- It's necessary to request the GPIO port numbers which need to used. The most readable way to do this, is to define an array of `gpio` structs and call the function `gpio_request_array`. (In the case of an error those need to be freed with `gpio_free_array`.)
- The high resolution timers, which are required for the time measurement, need to be initialized and the callback functions for each timer is set to `timer_irq`.
- The IRQs must be requested, so that the `irq_handler` can be called on an

interrupt. This is achieved by getting an IRQ number for every GPIO with the `gpio_to_irq` routine. (That part is architecture specific and not necessarly available on every ARM system. Fortunatley, it's available on the Raspberry Pi.) After getting the IRQ number, `request_irq` needs to be called, finally.

3.4 The IRQ Handler

After successfully initializing the GPIOs and IRQs, the only thing necessary to define is the interrupt callback function `irq_handler`. The function signature is:

```
1  irqreturn_t irq_handler(int
     irq, void *dev_id);
```

So, with our setup, the each key struct is associated with two irq numbers, one for the first and one for the second button. We can set the touch and the hit time time for each key when the handler is called with one of the associated IRQ numbers.

One important consideration here is, that we need to take care of possible button jittering manually. We solved this problem by calling another callback function which is delay with a high resolution timer. Until this second callback function is executed, we ignore every subsequent input on the current GPIO port, which seems to be a simple, yet reliable jitter resolution.

3.5 Button Events

After the timer callback function is called, the value on the corresponding GPIO port is read (which is in our case an integer, usually just 0 or 1). We can calculate the velocity from the saved timestamps for each button. This is done by calling the function `time_to_velocity` which interpolates the key hit velocity from a given interpolation function, and from given values for the minimum and maximum time difference. The interpolation function and the time thresholds can be set via `ioctl`. After calculating the velocity value, the `handle_button_event` routine is called, which sends an appropriate MIDI event, depending on the current value on the GPIO port and the calculated velocity.