CMIDID Kernel Driver & Linux AppleMIDI Documentation

Praktikum Linux & C - Technische Universität München

Felix Engelmann
felix.engelmann@tum.de

Michael Opitz
opitz@in.tum.de

Andreas Ruhland
ruhland@in.tum.de

Jannik Theiß
jannik.theiß@tum.de

August 3, 2014

1 Overview

1.1 Introduction

This document is split up into two main parts: The documentation for the *CMIDID* Linux kernel driver and the documentation for the *AppleMIDI* kernel driver. Development on both started in the summer 2014 during the *Linux- und C* lab at *TUM*. The original goal of this project was to build a Linux kernel driver which implements the *RPT MIDI* standard (as described in RFC 6295), yet after some initial discussion, the decision was made to focus development on the CMIDID driver, which enabled us to build custom MIDI hardware with low-cost devices like, in our case, the Raspberry Pi. Later during the development phase, after some promising progress on the CMIDID project, we started to implement a altered version of the RTP MIDI driver in the form of the *AppleMIDI* driver, which supports only a subset of the complete RTP MIDI standard, yet understands the Apple RTP MIDI session management - an extension of the standard.
The remainder of this document describes the main components and challenges of this project in detail. For a short overview of the implemented functionality and the possibilities available with CMIDID and Linux AppleMIDI, take a look at the project presentation in `/presentation/` and if you are looking for an introduction to build and test the drivers, read the `README.md`.

1.2 CMIDID Kernel Driver

The CMIDID Linux kernel driver can be used to build a fake MIDI keyboard by translating input on GPIO pins into arbitrary MIDI key events. For our demo setup, we used a Raspberry Pi with an attached breadboard to build several MIDI keys

via buttons, resistors and jumper cables. The CMIDID driver is highly customizable, and can, for example, use the input on two separate GPIOs to simulate a single MIDI key, with the added benefit, that the time difference between the GPIO events allows the calculation of the corresponding key hit velocity. With that in mind, it's possible to construct more advanced MIDI hardware compared to other projects with a similar scope (like, e.g. the *Makey Makey* controller @ http://makeymakey.com).

1.2.1 Authors

CMIDID was written by Michael Opitz, Andreas Ruhland and Jannik Theiß in collaboration.

1.2.2 Module Structure

The CMIDID module is split up into several components which was done in favor of good separation of concerns. The module is composed of three components: The main component, the GPIO component and the MIDI component. The main component handles mostly the setup and cleanup of the other components as well IOCTL. The MIDI component handles the creation and control of the corresponding virtual MIDI device and the GPIO component translates GPIO input into MIDI events. For a indepth documentation on this topic, take a look at `michael.pdf`.

1.2.3 Hardware

The CMIDID driver is intended to be run on a Raspberry Pi, which comes with several GPIO ports for simple hardware prototyping and which is able to run a Linux kernel as well as general purpose applications. In theory, every devices which supports Linux and has GPIO pins can be used to replicate our setup. An introduction on the challenges of compiling and running CMIDID on the Raspberry Pi can be found in `andreas.pdf`

1.2.4 Custom Keyboards

Custom MIDI keyboard key events are simulated by interpreting input on the GPIO ports as key hits and key releases. This allows flexibility for the construction of custom hardware, but has the downside of increased hardware prototyping time and more involved driver configuration.
An overview of our sample keyboard setup on a breadboard can be found in `jannik.pdf`

1.2.5 Module Configuration

The flexiblity of hardware development is probably the greatest advantage of the CMIDID module. To simply the process of MIDI keyboard construction, we added several ways to configure the module and the driver. It's possible to pass module parameters to define options which won't change during module runtime, like the number of available GPIO pins and the CMIDID driver can be configured during runtime via IOCTL from the userspace, which can be used for example to transpose the MIDI notes, i.e. add a constant to every note event.
The available configuration options are listed in `jannik.pdf`.

1.2.6 Virtual MIDI device

Besides the construction of the keyboard hardware, a virtual software based MIDI device is provided to communicate with other sound processing hardware

like for example a software synthesize like FluidSynth (http://www.fluidsynth.org) or TiMidity++ (http://timidity.sourceforge.net). Fortunately, the Linux kernel offers an interface to easily handle MIDI devices: the ALSA sequencer API. This API offers methods for MIDI device creation and manipulation and can be used from the userspace to route MIDI events between different devices.
Check out the the corresponding section to see how we used the ALSA sequencer API for this project: `andreas.pdf`

1.2.7 GPIO Handling

The GPIO component of CMIDID module handles incoming triggers on the GPIO pins. Depending on the configuration, rising edge and falling edge events are interpreted as MIDI key hits and subsequent releases. Those signals are translated into a specific MIDI event: Either noteon or noteoff. Additionally, the GPIO component is responsible for the calculation of the velocity for the noteon events. This is done by measuring the time difference between the virtual key hits and interpolating the velocity according to a given interpolation function.
For more details on the handling of GPIO input, see `michael.pdf`.

1.3 AppleMIDI

1.3.1 Author

The AppleMIDI kernel driver was exclusively developed by Felix Engelmann.

1.3.2 Information

The AppleMIDI driver project was forked off the cmidid driver, as it might be useful for other projects too. The alsa sequencer core was considered a good interface, as it is low level enough for performance but easily controllable. Another advantage of two separate driver is, that when synthesizing directly on the local machine, there are no network security issues.
The driver acts as a one-directional bridge for MIDI events from the alsa core to network clients. For transport, it uses an extension of RTPMIDI (RFC 6295)) which is called AppleMIDI. There, the session management is included into the RTPMIDI. The foundation of the code is the midikit project, from which all libc independent functions were used.
In the current implementation, there are several drawbacks from an optimal bridge. It does not accumulate multiple events which are issued in a short interval. Also only a subset of all MIDI commands is implemented for binary encoding. A backward correcting journal might be added too. The ideal functionality will provide both directions, so MIDI commands will be transmitted and received.
For live performances, this module has an average processing time of several hundred microseconds, which is under the threshold of recognition.