RATT - Relatively Accurate TurnTable

Relative turntable motion detection through mouse sensors

Zoppi Andrea
Matr. 765662, (andrea.zoppi@mail.polimi.it)

Report for the master course of Embedded Systems
Reviser: PhD. Patrick Bellasi (bellasi@elet.polimi.it)

Received: <MONTH>, <DAY> 2012

Abstract

Cheap jogwheel encoders for emulated DJ turntables are often inaccurate, due to the low CPR in the order of some tens. Precise encoders are rather expensive and are not convenient to keep track of very fast rotations, because they are too much accurate for the purpose.
The presented research tries to improve the detection of at least small and slow relative rotations of the jogwheel, by employing a cheap COTS mouse sensor, and keep the absolute position or fast rotation with the classic cheap optical encoder.
A very crude HID demoboard was developed, so that some simple tests were done. An extended proposal is also described, in order to achieve better performance and more features, to match those of a commercial DJ controller.

1 Introduction

In the latest years, the market of digital-DJ related products has grown considerably. By the way, cheap digital turntable emulation is still tricky, because the design of a cheap yet accurate jogwheel is still a challenge even with the technology available today.
It is true that state-of-the-art processing units are very fast, but there are still issues such as those related to protocol latency, precise and fast plate motion detection, motion samples interpolation, and so on.
These issues are not a big problem for the average DJ, but they arise when requiring a higher performance (e.g. *scratch*) while keeping the costs low.
The proposed approach is based on COTS components called *optical motion/mouse sensors*, which can provide a very good accuracy when detecting small local motions, which is a behavior difficult to obtain with cheap encoders.
A simplified verison of the proposal was developed on a crude prototype, just to check if it is worth at least for the average DJ – the most demanding ones do not care about the price of products, and still rely on timecoded vinyl emulation even tough high CPR optical encoders are available at the same overall price.

2 Current market

The common commercial approaches can be divided into two groups: jogwheels based on optical encoders, and reuse of vinyl turntables (or CD players) as if they were digital jogwheels. These two technologies will be described in the following, showing their pros and cons.

There exist also some other ways to emulate turntables, which are currently still in a niche. For example, there are some touchscreen-based [1] [2] or capacitive [3] controllers, which follow the market wave of touchscreen devices.

There are also some evolutions of the optical jogwheels, which are motorized [4] [5] and thus more suitable for professionals, but rather expensive.

2.1 Optical encoder jogwheel controllers

The most common technology for turntable emulation is based on optical encoders. An optical encoder is a device which detects motion by counting the number of steps an evenly-marked wheel performs. It is found in almost all purely digital DJ *controllers*, which in this context are referred to those remote digital devices used by the DJ to control the user application. Some examples of commercial controllers with jogwheels can be found in [6] [7] [8].

Common jogwheels have a resolution (*CPR, Counts Per Revolution*) in the order of tens, thus are not suitable for *scratching*, and are usually addressed only in coarse track navigation, or *bending*. Even a resolution in the order of some hundreds cannot be enough for scratch or precise motion tracking. For example, with a 720 CPR encoder it is possible to detect only motions of half degrees, that for a 12 inches wide wheel is still low – keep in mind that a vinyl spins at roughly 150 degrees per second, and good sampling should require at least 1500 samples per second to track it decently enough at nominal speed.

System architecture The basic architecture of these controllers is shown is **Figure 1**. The controller commonly has a set of input devices – buttons, knobs, sliders, *etc.* – so that the user can *map* these inputs to some software actions, such as the play/stop events, or the desired volume level.

A special kind of input device focused throughout this work is the optical encoder, which will be described in depth later.

Commonly, there is a also set of output devices – LEDs, displays – so that the user's sight should not always keep switching switched between the computer monitor and the controller to see what is going on.

All these devices are managed by a MCU, which detects their changes, and generates meaningful messages to be sent to the DJ software, or receives messages from the latter.

The communication between the controller and the software is often performed through an USB bus with HID/USB or MIDI/USB protocols, but some controllers still rely on the plain old MIDI port (see Section 4).

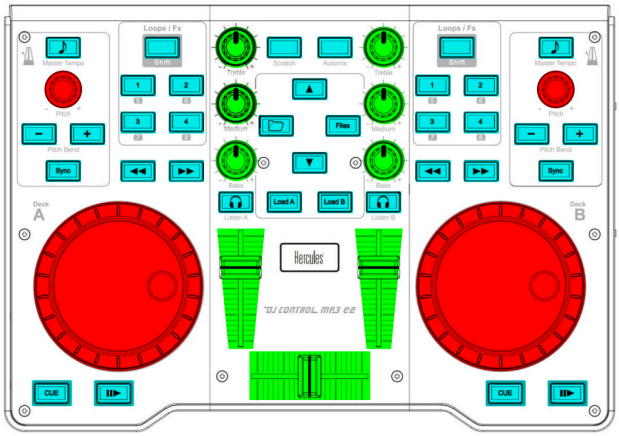

Figure 1: *Hercules DJ Control MP3 e2* [9] intereface schematic with buttons, incremental encoders, sliders and knobs

Wheel architecture The wheel is emulated with a so-called *jogwheel*. It is a disc whose full rotation is divided into equally-spaced angle slices. Each slice is assigned a code.

Usually, the code is marked on the wheel with holes aligned on circles (see **Figure 2**), so that holes can be detected by light detectors mounted on the chassis. The light detector is almost always made with a LED which points towards the disc, and on the other side the light is detected by a fast phototransistor. These light sensors are positioned so that they can detect one and only one code per slice.

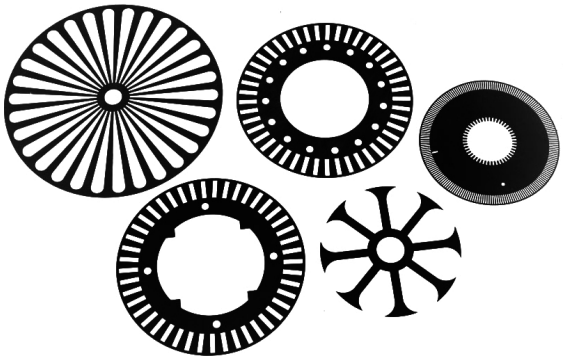

Figure 2: A set of incremental (quadrature) rotary encoder wheels

The code is either abolute or relative. Absolute encoders assign a unique code to each slice, so that it is always possible to know the current wheel angle by just reading the light detector outputs. Due to the need to have a high number of bits, the number of holes can also grow exponentially (usually as the power of two), and the production of precisely aligned marks and sensors is expensive – misaligned ones can provide misdetections of the angle, even with robust codes such as the *Gray code*.

Instead, relative encoders just need the two least significant bits of an absolute code, thus cheaper to manufacture. On the other hand, it is not possible to know the absolute rotation without any additional bits. This is why there is often a mark which signals a full revolution been performed, and needs an additional flag bit. The particular subset of the Grey code used for the relative motion detection is called *quadrature code*, because only 4 code sequences (phases) can be generated by moving to the adjacent wheel slice as seen in **Figure 3**.

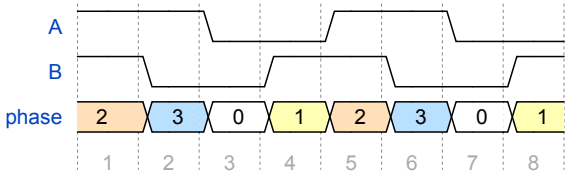

Figure 3: Quadrature pattern, going forward left-to-right

An example of jogwheel internals can be seen in **Figure 4**. It is a close-up of the *Vestax VCI-400* [8] jogwheel architecture, with a high-resolution wheel – segments can be barely seen – and an *Agilent HEDS-9700* quadrature encoder.

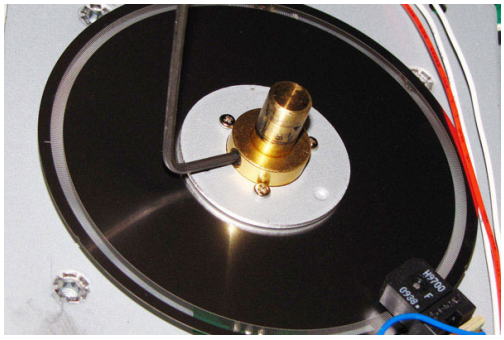

Figure 4: A *Vestax VCI-400* jogwheel being disassembled

Motion detection When the user turns the wheel, the light detectors can convert the sight of light into the code assigned to the focused disc slice. The digital code is then triggered by the MCU through some interrupts, and a message containing the motion (or even the absolute angle) is sent to the user software.

Pros

- Easy to manufacture
- Code detection is inherently digital
- Fast code transitions can be processed easily
- A cheap MCU can handle jogwheels as well as all the other digital devices commonly found in DJ controllers

Cons

- Small motions have poor resolution with cheap encoders
- High resolution encoders are too much expensive for the purpose

2.2 Timecoded media turntable emulation

An alternative way to emulate a turntable in software is to use a *timecoded audio track*, which is an audio stream coded so that the software can read the track position just by decoding the incoming audio stream.
This technique makes it possible to use existing turntables or CD players to control the user software, which in turn will emulate the turntable behavior.
The good side of this approach is that a DJ, who already owns turntables or CD players, can keep using them just by buying a sound card with the appropriate audio inputs. This way the DJ can have almost perfectly the same old feeling, because he is still using the same equipment.
On the bad side, vinyls and CDs are very sensible to usage, and decay easily. This makes the timecode unreadable in the ruined parts of the support, where software cannot always understand the code thus producing jittered or jerky behavior.

In addition, turntable needles must follow tracks almost perfectly, or the timecoded signal would degradate at the ADC side, especially the phase component which is necessary for the purpose, but almost ignored in audio players since the human ear has poor phase sensitivity.

Another bad point relates to the overall performance. It is true that with this technique the performance is almost the same of a real vinyl, but the need of an intermiate sound card, which in turn is often connected through the USB bus, just makes low latencies hard to achieve, unless the host computer is powerful and well optimised to reach soft-realtime requirements.

With a timecoded media is only possible to control the track position (phase) and pitch (frequency), whihc is good for plain turntable emulation, but it is impossible to use some features – effects, precise loops, track preview, etc. – of some professional CD players.

When the disc spins at low speed, the intrinsic high-pass filter of needles and soundcards will fade the signal, and it can become difficult to find the zero-crossings while decoding. This issue makes slow scartches difficult to emulate because of jitter and corruption, and will be mitigated with the work devolped in the rest of this document.

Finally, a novice DJ would hardly choose this approach, because the overall price of the equipment can be rather high – turntables/players + good soundcard + accurate needles + cables + hi-performance computer can easily exceed $3000.

System architecture A common commercial architecture [10] [11] can be seen in **Figure 5**. The existing turntables or CD players are connected to the appropriate soundcard inputs.

The soundcard can be placed either inside or outside the host computer. Internal soundcards are very fast in transfering data from the incoming audio signal to the CPU, thanks to the fast system bus (PCI or PCI-E).

However, as internal soundacards are almost always designed for desktop computers, while notebook computers are much easier to carry, the choice of an external soundcard is the very most common.

External soundcards are mostly connected through an USB bus, but some professional FireWire soundcards are still on the market. The USB bus has the drawback of having a fixed minimum latency of 1 ms for isochronous signals, which increases latencies even more, while FireWire is faster – good internal soundcards have a negligible latency.

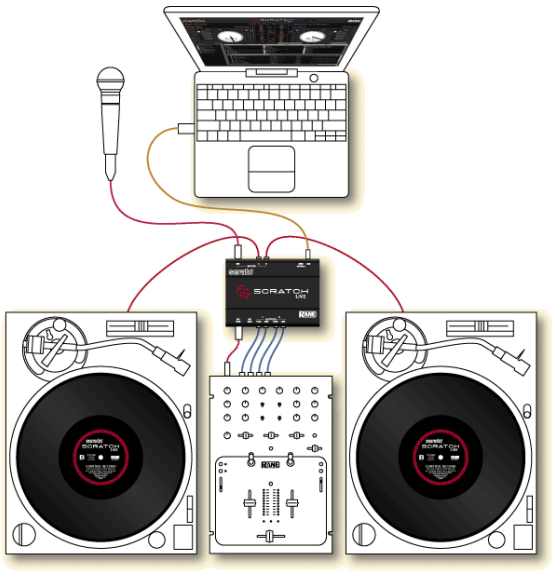

Figure 5: *Serato Scratch Live* setup with two turntables

Signal pattern The timecoded signal is printed on the source media – vinyl, CD – with patterns recognized by the specific user software. In general, the timecoded signal is composed by a left sine and right cosine signals at a constant frequency, usually in the range of 1 kHz to 3 kHz. The amplitude of each half-wave of the same sign is slightly modulated with a proprietary digital code, which represents the absolute position inside the whole timecoded track.
In **Figure 6** it is possible to see a slice of the *Serato Scratch Live* [10] timecode signal (aka *noisemap*), with some amplitude-modulated bits at the sine wave top peaks.

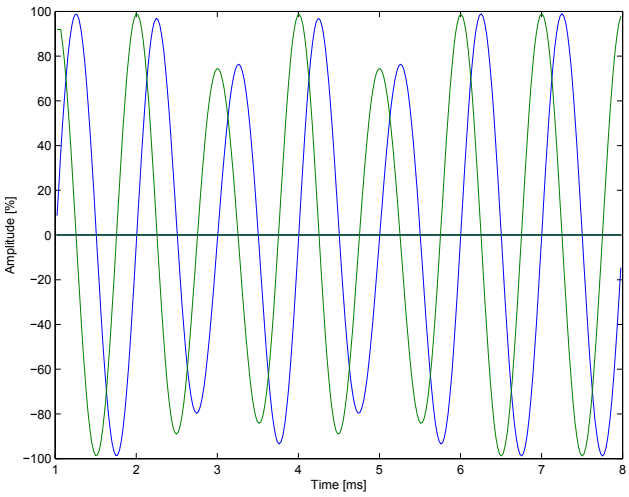

Figure 6: Slice of the *Serato Scratch Live* timecode

To read the code from a timecoded stream, it is sufficient to trigger a positive-slope zero-crossing on one channel, and read the amplitude on the other channel (peak), then convert the amplitude into a bit of the code word. If the absolute position is not needed, instead of the whole code it is possible to interpret the sine/cosine simply as a quadrature control signal, for relative motions.
The code is designed to be decoded in both directions, so that the user can navigate the emulated turntable just like a real vinyl. The most advanced codes are designed to support fast error correction, because dust, decay of the vinyl, or even EMF, would generate a corrupted signal.
In fact, if a single code is a bare label of the position, for example of 20 bits, it is necessary to read at least 20 consecutive half-waves. With a nominal frequency of 1 kHz, the minimum latency would become 20 ms, which is rather high for audio manipulation by a human. Advanced codes would still correctly read the first 20 half-waves, but then each subsequent bit is sufficient to get the adjacent code, with only 1 ms delay at 1 kHz.
Some problems arise when the turntable is turning slowly. Due to the bandpass behavior of the needle and active input/output stages of the whole signal chain, slow transitions have a small amplitude. As it is well known in communication technology field, slow and small transitions make zero-crossings much harder to be tracked correctly, thus generating jitter or missing codes. This is where optical motion sensors show their best performance instead.

Pros

- Same old equipment
- Just add soundcard to a computer
- Natural feeling and performance

Cons

- Needs high quality components in the signal chain
- Expensive for a novice
- Signal subsceptible to corruption
- Bad tracking at low speeds
- Expensive signal processing

3 Optical motion sensors

Optical motion sensors are commonly found in computer pointing devices called *mouses*. The job of such sensors is to capture a *photograph* (i.e. a *frame*) of what is beneath, compare it with the previous one, and finally compute the motion (i.e. the distance) from the previous relative position.
Their main strength is the achievement of very high resolution comparisons, in the order of hundreds, if not thousands, *Dots Per Inch* (*DPI*), at a very low price – less than a dollar for average sensors, or a few dollars for high performance (*gaming*) mouse sensors. Also, the maximum detectable speed is in the order of some tens inches per second, compatible with human and motion.
As a drawback, the absolute accuracy is poor, as they are designed to capture the slow relative movements of a screen cursor. It can be repositioned by the software, or

its relative motion is exploited for some games (e.g. *First Person Shooters*), thus making absolute motion tracking clueless.
As can be inferred from the previous statements, mouse sensors come in different classes, based on their main applications. The most common sensors are for office or home use, and provide average accuracy and speed. Other sensors are designed for wireless devices, and can provide automatic switching between average (or high) and low performance, based on the amount of interaction of the user in the last time window. The most advanced sensors are for those professionals or enthusiasts who need very high speed and precise motion detection, for example for pro-gamers.

3.1 Architecture and operation

A generic mouse sensor is composed by components of various nature, as seen in **Figure 7**.

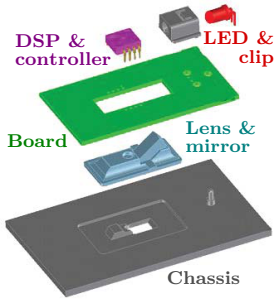

Figure 7: Common motion sensor assembly

Light source A light source, typically a LED, or a laser for the most advanced models, illuminates the small surface to be captured.

Lens A lens focuses the surface to get the best resolution at detector side. The lens also increases the effectiveness of the light source on the surface, when correctly focused on the surface beneath through a mirror.
Usually the lens is placed at a few millimiters from the surface, and it is quite important to keep the nominal distance for the best perofrmance.

CCD The actual light sensor is a common *Charge-Coupled Device* (*CCD*). It is designed to be very fast, to achieve a high frame rate. It is also sensitive to a very small light spectrum, e.g. the infrared one, so that environmental light noise is rejected, and image artifacts are acquired better.
There are also some considerations based on the light type. LED-based devices work well with all the bumpy surfaces, even the dark ones, but cannot work on trasparent ones, where the reflection is very bad. They are suited for everyday use. Instead, laser-based give a way better tracking on all surfaces but the dark ones. The tracking performance is suited for professionals.

DSP The frame impressed on the CCD is then processed by an *ad-hoc DSP*, which computes the distance – in (x, y) frame coordinates – from the previously acquired frame. The DSP must be fast enough to reach the 10x-in/s maximum detected velocity in both directions.

Controller Finally, the controller accumulates the DSP deltas. Some configuration parameters and device properties, along with deltas, can be accessed through an interface to an external controller.

3.2 Communication

Optical motion sensors communicate with an external processor through an interface which often belongs to standard I/F types, even though some pins are frequently added for faster or ad-hoc operation – chip shutdown, chip selection, flow control, and so on.

Quadrature Sensors with only quadrature wave outputs (see **Figure 3**) are now outdated, because they do not offer any advanced features, and require an interrupt-driven counter at controller side. On the other hand, quadrature outputs can be used with any device which natively accepts such signals.

SSP When requiring some more flexibility, it is possible to adopt a *Synchronous Serial Port* in both *Serial Peripheral Interface* (*SPI*) or *3-wire*, respectively with full-duplex and half-duplex capabilites, to communicate with motion sensors.
They usually do not require handshaking, even though some control pins are often used in such a way (e.g. the common *shutdown* pin). Also, the bit rate is often high enough for simple and fast communication – from 1 MHz to some tens.
Due to its message-based nature, it is possible not only to read internal counters, but also to get and set configuration parameters of the target device.
They can often support multi-master/multi-slave topologies, even though the most common is single-master/multi-slave. Slaves are usually chosen by a demultiplexed selection signal, one per slave.
An example of *write* operation over a 3-wire SSP bus is shown in **Figure 8**. The single master device signals to the single slave an incoming *write* operation, tells the register address and its value.
An example of *read* operation over the same hardware is shown in **Figure 9**. The master signals an incoming *read* operation and tells the address. After that, it lets the slave drive the SDIO line, then eventually reads the addressed register value.

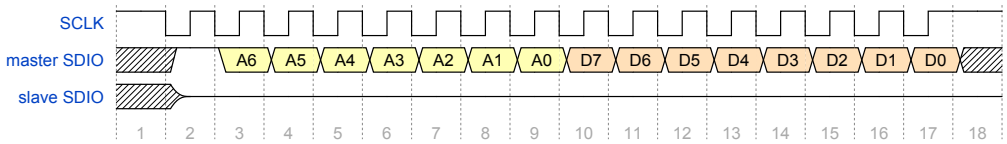

Figure 8: A typical write operation over a half-duplex, 3-wire SSP

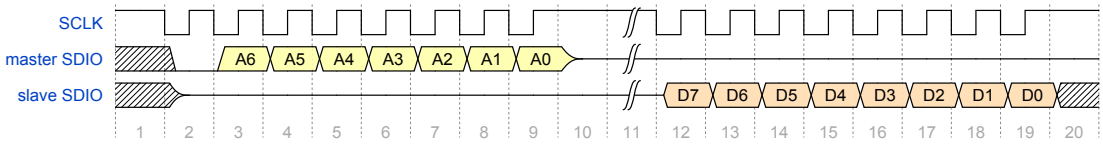

Figure 9: A typical read operation over a half-duplex, 3-wire SSP

4 Controller communication protocol comparison

When developing a music-oriented controller, there is always a debate on which communication standard is the most suitable for the application. The most common standard are described in the following.

4.1 Plain MIDI

The truly *de-facto* standard in music-oriented communication is the *Musical Instrument Digital Interface* (*MIDI*). This was developed in the '80s to be easy and cheap to manufacture, robust and rather complete for standard music production, with some degree of freedom for sub-protocols developed by manufacturers. It is still widely supported by digital music equipment nowadays.
Common output and input circuits can be seen respectively in **Figure 10** and **Figure 11**. Basically, it is a standard serial point-to-point connection based on a common UART powered at TTL levels. Data transmitted by the UART is converted into current bursts, which light the optocoupler at the receiver. The optocoupler provides galvanic isolation, so that no *current loops* can create audible noise in the target device, which used to be a synthesizer in the first place.
Each MIDI connection can handle up to 16 *channels*, i.e. virtual music devices. This is enough for common synthesizers, but not to drive an entire production studio.
The event-based nature of the protocol makes it suitable for most of the situations, but for events which could be accumulated, or state streaming. For example, jogwheels could generate way too many motion delta events, usually as Control Change messages, which would easily flood the entire bus. A packed streaming state would be better suited in such cases. Also, when transmitting to a host computer, its operating system (often time-shared), software buffers can saturate while processing too many events at a time, or complex sound textures.
The protocol does not support any kind of flow control. Massages can be lost, partially received (ignored), contain errors, without being corrected or asked for retransmission. Also, the relatively slow baud rate can generate small delays between sound generation/control events, which can be detected by the human brain, which is very sensitive to audio timing.

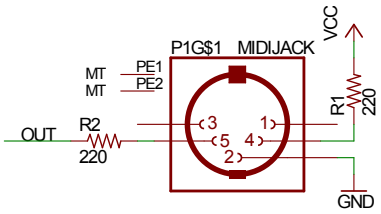

Figure 10: MIDI output circuit, UART sends *OUT* signal

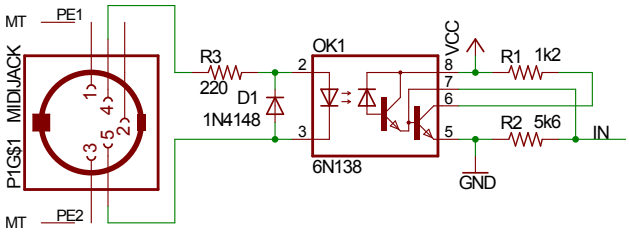

Figure 11: MIDI input circuit, UART receives *IN* signal

Basic protocol The serial protocol message word is 8 bits long, with one start (low) and one stop (high) bits, MSb first, 31250 baud. The protocol is message-driven, with standard messages 3 bytes long. Special messages are 1 to 3 bytes long, while manufacturer-defined *System Exclusive* (*SysEx*) messages can be arbitrarily long. Most of the messages were associated to common synthesizer features [12], even tough manufacturers often interpret them freely (e.g. through *remapping*).
The first word of a message (control word) is always identified by the MSb set to 1, while it is always 0 for data words, which are always 7 bits wide.
Standard messages are those referred to the actual music content, such as *Note On/Off*, *Control Change*, *Pitch Change* and so on, also called *voice messages*. A timing diagram of a voice message is shown in **Figure 12**.
The control word identifies the message type and the chan-

nel number. The second word is usually the identifier of the control (key, knob, slider, wheel, etc.) being actioned, and the third word tells its value or velocity.
Special messages include transport control (time, position), program selection, system messages, etc. These are advanced messages, often only partially supported, if not at all, by cheap controllers.

Pros

- *De-facto* standard for digital music equipment
- Intuitive message semantics
- Simple, robust, noise-free, cheap hardware

Cons

- Slow communication
- No flow control
- Low resolution controls
- No streaming state support

4.2 MIDI/USB

The *Universal Serial Bus* (*USB*) [13] is probabily the most available for consumer electronics interfacing nowadays. Its multipurpose nature, speed, and robustness, made it the *de-facto* standard for connecting an actual *universe* of devices. Among this huge load of devices, also the digital music related ones can be found.
Since the MIDI standard was the most common digital music standard at the time USB was released, it was convenient to encapsulate it inside USB packets. So, an *application protocol* was developed over the USB protocol: the *MIDI over USB* (*MIDI/USB*).
This way, compatibility with standard MIDI interfaces was kept. Production software and hardware devices kept using it, an so did music production people. USB is simply the low-level interface, which can be found on all consumer computers.
MIDI/USB has some advantages over the plain MIDI protocol. Since USB has a higher baud rate than MIDI, it is possible to reduce delays when handling many messages in a short interval.
Also, MIDI/USB introduces the concept of *cable*. It supports up to 16 cables, which are 16 virtual plain MIDI devices. This makes theoretically possible to drive up to 16 hardware synthesizers through a single USB cable, by dispatching the 16 virtual cables to as many real plain MIDI devices.
The MIDI/USB protocol exploits the features of *bulk* transfers. This way, packets will not be lost, thanks to the flow control of USB bulk transfers.
USB devices can also show many indipendent behaviors, by choosing different application protocols per each USB endpoint. For example, this allows to have a MIDI/USB device with an integrated HID/USB trackpad, by using a single USB cable, like the *M-Audio Torq Xponent* [14].

All the MIDI drawbacks are kept, except for the baud rate and flow control. The USB, by its time-shared nature, always introduces some delays – at least 1 ms per query, and a few milliseconds for software stack processing. As a drawback of bulk transfers, the delivering time is uncertain, because many retansmissions can be issued, or there is not enough free bandwidth on the bus, introducing even more delay.

Basic protocol The MIDI/USB protocol is a *sub-class* of the *Audio class*, defined by the USB standard [13]. This parent class is oriented to audio equipment communication and control – speakers, microphones, keyboards, controllers, synthesizers, DSPs can all be driven by this class.
The packet is composed by a *header byte*, and the remaining 3 bytes are the encapsulated MIDI packet – all but the SysEx messages, which have variable length. The header byte addresses the cable, and defines the following MIDI message type – some bytes can be unused, and thus ignored by the USB parser. SysEx messages are simply split into 3-bytes chunks, merged by the USB parser when the trailing byte(s) are received.
The protocol supports *bulk* USB transfers. They add flow control to bare USB streams, which in order adds flow control to the encapsulated MIDI stream. Bulk packets have the lowest priority over the USB bus, which could introduce delays when the bus is saturated by *interrupt* and *isochronous* endpoint transfers. Also, packet handshaking and retransmission may add some delay too.

Pros

- USB is *de-facto* standard for consumer electronics
- Much higher bandwidth than plain MIDI
- Up to 16 virtual cables
- Multi-purpose device through a single USB cable

Cons

- All the MIDI drawbacks, except for baud rate and flow control
- Delays introduced by USB time-sharing and bulk transfers

4.3 HID/USB

Besides MIDI/USB, another application protocol was developed to handle those devices which interface with human beings, and is much widely implemented. The *Human Interface Device* (*HID*) [15], in fact, is an USB class with huge flexibility and support – it can be found in almost any USB device with keys, knobs, sliders, simple displays, LEDs, control wheels, and so on.
USB endpoints for HID applications are scheduled with *interrupt* priority. This means that HID is suitable for low-latency devices, but does not offer high bandwidth.

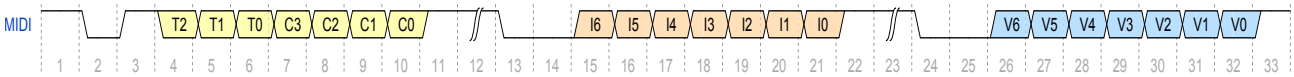

Figure 12: Timing diagram of a MIDI voice message

A strength of the HID protocol is that it is highly available. Any decent host OS has full native support for it. Also, HID reports are described upon connection, and the host can actually parse not only the bare syntax, but also its semantics for some common cases – keyboards, joysticks, gloves, *etc*. This often makes the device immediately available to the user, without the need of custom device drivers.

Being developed over the USB protocol, HID packets suffer from USB scheduling delays. This means that a *USB full speed* device has a minimum latency of 1 ms between its packets – the minimum *interrupt* latency.

Basic protocol The HID protocol is based on the exchange of *report* packets. A report has a structure described in the setup phase of the USB HID class interface. Its size is constrained by that of USB *interrupt* packets, so it cannot contain too much data. By the way, it is big enough to support *state streaming* of a small set of data, or to describe a few events per packet.

State streaming becomes handy when tracking the status of some critical or rapidly changing data sent by the device. In the case of the simplified RATT prototype (see Section 5), deltas and absolute positions of both the mouse sensor and the incremental encoder are continuously streamed. This optimizes the USB bandwidth usage. In fact, those devices would generate a huge number of motion events if sent with an event-based protocol, like MIDI/USB, wasting the available bandwidth and introducing queuing delays.

The exchange of HID reports is straightforward, but the report definitions need some attention. As told before, the HID report structure must be communicated when configuring the HID connection, with a *Report* descriptor. This descriptor is written in a dedicated language [16], which might be tricky – some tools are available to help the design of such descriptors.

Pros

- Natively supported by most operating systems
- USB *interrupt* priority
- Suitable for small state streaming
- Common semantics automatically inferred

Cons

- Sill some USB delays
- Custom HID reports to design

5 Simplified proposed approach

The proposed approach was developed so that it is possible to achieve good performance both at high and low speeds, without introducing expensive parts.

The main idea is to use a mouse sensor to measure slow local motions, which are difficult to process with timecode, and need high resolution jogwheels. As seen in the previous section, this is not possible with cheap designs.

Slow local motions are common when scratching or trying to reposition the virtual needle of the emulated turntable, for example when the DJ is searching for a good point where to start playing from (*cue* point).

When the disc is spinning at cruise speed, or when the rotation is fast enough, the classic methods can be used to keep track of the position. Mouse sensors, in fact, are not able to keep track of absolute movements, especially when they are very fast. Anyway, even cheap sensors have a resolution in the order of several hundreds, if not thousands, DPIs nowadays.

As proven later, the disc can theoretically be divided in thousands slices, thus providing a local resolution higher than the most expensive encoders on the market – obviously in the set of those affordable for a DJ, not the state-of-the-art encoders for hi-end industrial machineries.

In the following, a simplified prototype will be presented. A low-end MCU and a standard COTS mouse sensor were chosen, just to see if it is possible to achieve good performance with a simple circuit.

5.1 Hardware architecture

The simplified prototype has a very crude hardware architecture. It is split into three modules: the *controller board* (aka *main board*), the *sensor board* and an optional *timecode preamp board* for the use with timecoded media. This subdivision was done just to decouple the controller with the sensor, for further experiments with other hardware configurations.

Controller board The controller board in **Figure 13** hosts the MCU, a *PIC18LF14K50* [17] by *Microchip*, which is a 8-bits MCU running at 48 MHz (16 MIPS). It is fast enough to handle a single wheel, but nothing more advanced such as DSP, which is left to the user software application.

This board also mounts the voltage regulator, to get 3.3 V from the standard 5 V sourced by the USB host. Bulk capacitors keep it stable.

There are also a user button for input, and three LEDs for user interactions.

A standard UART can be used for basic messaging with an optional text console.
There are also two pins dedicated for an additional quadrature encoder input. They can be configured to be directly coupled with the MCU comparator interrupts, in order to trigger encoder changes.
To communicate with the sensor module some control and interrupt signals, and a half-duplex serial port are provided.
The USB bus is directly connected to the MCU, which has an internal *Serial Interface Engine* configured for *Full speed* transfers.

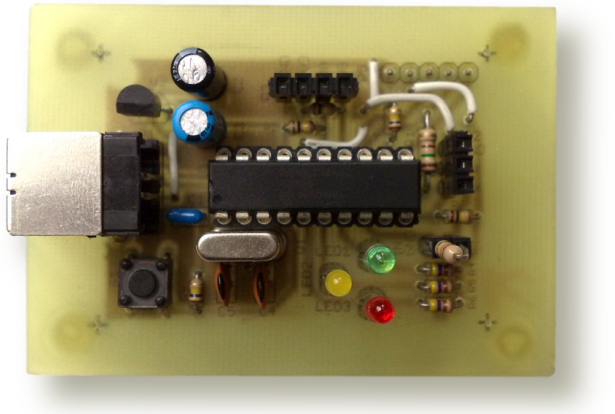

Figure 13: The controller prototype board

Sensor board The sensor board in **Figure 14** simply hosts the mouse sensor, the surface illumination LED, and the lens. The sensor is an *ADNS-2080* [18] by *Avago*. It is a sensor aimed at office users, with average performance and low cost. As for the MCU, higher performance modules are on the market, but the challenge of this proposal was to find a basic average solution, which can be improved with further research. Briefly, the sensor can reach an interesting resolution of up to 2000 DPI, and motion speed up to 30 in/s (76.2 cm/s). There is no guarantee of constant latencies, because the clock speed is variable.

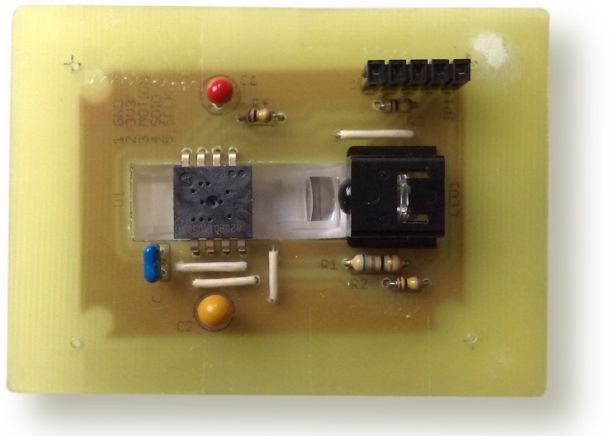

Figure 14: The optical motion sensor prototype board

Timecode preamp board An optional board, seen in **Figure 15**, was designed to properly amplify a *Line* – no *Phono*! – timecode signal so that it can be recognized by comparators of the controller board.
Each audio channel can be amplified in amplitude, by controlling the inverting gain of the opamp with a potentiometer. A spare opamp generates the $V_{DD}/2$ voltage reference for single-supply conditioning.
Two LEDs help the user in keeping the overall timecode level compatible with the comparator inputs on the controller board, by warning if the signal is near saturation, or too low to be squared correctly by comparators. In fact, when the volume is too low or saturates, the sinusoidal timecode signals won't be squared with 50% duty cycle, which can generate some jitter.

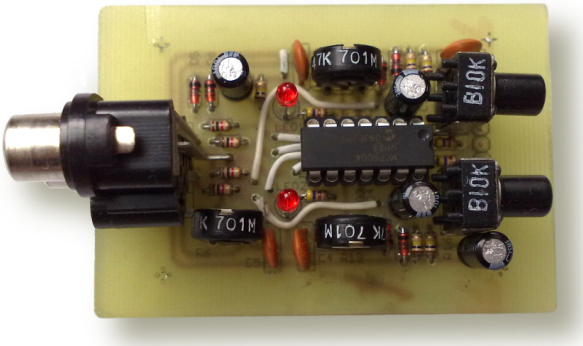

Figure 15: The timecode preamplifier prototype board

5.2 Firmware architecture

Due to the lack of a *Real-Time Operating System* (*RTOS*) support on the chosen MCU, the application is written in a *while-loop* fashion, with *interrupt-driven* tasks at *high priority*.
A logical view of the implemented firmware framework is depicted in **Figure 16**.

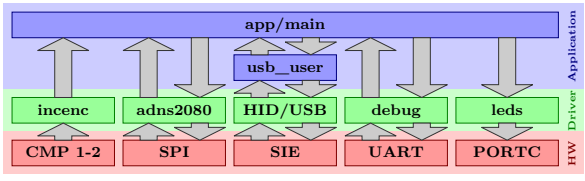

Figure 16: Logical view of the simplified firmware architecture

Bootloader In order to simplify firmware deployment, a HID bootloader is added to the development prototype. This fills the first 2048 words of the program space, which is not that small. The bootloader is the one provided by

the *Microchip Application Libraries* [19], and implements HID control.

The bootloader can be called by plugging RATT into the USB host socket, while pressing the user button. The developer can then use the HID bootloader software provided by Microchip to download the complied HEX executable to the program memory of the MCU.

HID/USB module The HID/USB module is devloped over the *HID Simple Custom Demo* provided with the *Microchip Application Libraries* [19], using the *USB Framework* version *2.9d*. The code itself is almost the same, so please refer to the documentation delivered by Microchip for more information, as the topic is very complex and cannot be described in a few words here.

RATT is a simple I/O HID device, which uses the endpoint 1 for both input and output transfers. In the following, the descriptors used by HID/USB will be described briefly.

Table 1 shows the *Device* descriptor. It tells the host it is a standard USB 2.0 device, with its vendor and product identifiers (dummy in this case), with only one configuration. It also indicates there are a manufacturer and product name strings to be received and indexed later.

Offset	Field	Size	Type	Value
0	bLength	1	Number	18
1	bDescriptorType	1	Constant	0x01
2	bcdUSB	2	BCD	0x0200
4	bDeviceClass	1	Class	0x00
5	bDeviceSubClass	1	SubClass	0x00
6	bDeviceProtocol	1	Protocol	0x00
7	bMaxPacketSize	1	Number	8
8	idVendor	2	ID	0xDEAD
10	idProduct	2	ID	0xBEEF
12	bcdDevice	2	BCD	0x0002
14	iManufacturer	1	Index	1
15	iProduct	1	Index	2
16	iSerialNumber	1	Index	0
17	bNumConfigurations	1	Integer	1

Table 1: USB *Device* descriptor

The single *Configuration* descriptor in **Table 2** tells the host that the device is powered by the USB bus at 100 mA maximum, and there is only one interface for this configuration.

Offset	Field	Size	Type	Value
0	bLength	1	Number	9
1	bDescriptorType	1	Constant	0x02
2	wTotalLength	2	Number	41
4	bNumInterfaces	1	Number	1
5	bConfigurationValue	1	Number	1
6	iConfiguration	1	Index	0
7	bmAttributes	1	Bitmap	0b11000000
8	bMaxPower	1	mA/2	50

Table 2: USB *Configuration* descriptor

The single *Interface* descriptor in **Table 3** simply indicates that it is a HID device (HID class). All the other options are ignored for this device.

Offset	Field	Size	Type	Value
0	bLength	1	Number	9
1	bDescriptorType	1	Constant	0x04
2	bInterfaceNumber	1	Number	0
3	bAlternateSetting	1	Number	0
4	bNumEndpoints	1	Number	2
5	bInterfaceClass	1	Class	0x03
6	bInterfaceSubClass	1	SubClass	0
7	bInterfaceProtocol	1	Protocol	0
8	iInterface	1	Index	0

Table 3: USB *Interface* descriptor

The *HID Class-specific* descriptor in **Table 4** is for the HID 1.11 protocol, no country-specific address, with a single HID report descriptor 48 bytes long.

Offset	Field	Size	Type	Value
0	bLength	1	Number	9
1	bDescriptorType	1	Constant	0x21
2	bcdHID	2	BCD	0x0111
4	bCountryCode	1	Number	0x00
5	bNumDescriptors	1	Number	1
6	bDescriptorType	1	Constant	0x22
7	wDescriptorLength	2	Number	48

Table 4: HID *Class-specific* descriptor

The USB *Endpoint* descriptors in **Table 5** and **Table 6** refer to the bidirectional endpoint 1, used to handle HID transfers, polled each 1 ms with *interrupt* scheduling. Since the user needs very fast responses for very few data from the motion sensor (16 bytes per report), these settings are optimal for the device being developed.

Offset	Field	Size	Type	Value
0	bLength	1	Number	7
1	bDescriptorType	1	Constant	0x05
2	bEndpointAddress	1	Number	0x81
4	bmAttributes	1	Bitmap	0b00000011
5	wMaxPacketSize	2	Number	16
6	bInterval	1	ms	1

Table 5: USB IN *Endpoint* 1 descriptor

Offset	Field	Size	Type	Value
0	bLength	1	Number	7
1	bDescriptorType	1	Constant	0x05
2	bEndpointAddress	1	Number	0x01
4	bmAttributes	1	Bitmap	0b00000011
5	wMaxPacketSize	2	Number	16
6	bInterval	1	ms	1

Table 6: USB OUT *Endpoint* 1 descriptor

As told before, manufacturer and product names are indexed inside the Device descriptor. The language code and those names are defined by the *String* descriptors shown in **Table 7**, **Table 8**, and **Table 9**.

Offset	Field	Size	Type	Value
0	bLength	1	Number	4
1	bDescriptorType	1	Constant	0x03
2	wString	2	Unicode	0x0409

Table 7: USB *Language Code* string descriptor

Offset	Field	Size	Type	Value
0	bLength	1	Number	12
1	bDescriptorType	1	Constant	0x03
2	wString	10	Unicode	"TexZK"

Table 8: USB *Manufacturer* string descriptor

Offset	Field	Size	Type	Value
0	bLength	1	Number	10
1	bDescriptorType	1	Constant	0x03
2	wString	8	Unicode	"RATT"

Table 9: USB *Product* string descriptor

Finally, the HID *Report* descriptor in **Table 10** describes the syntax and semantics of the actual data being transferred by the device application. Because of the custom semantics, the descriptor simply indicates 16 bytes for both input and output transfers. Semantics of the *IN* (device-to-host) report are listed in **Figure 17**, while that of the *OUT* report is currently reserved.

Field		Value	
0x06	Usage Page	0xFF00	Vendor Defined Page 1
0x09	Usage	0x01	Vendor Usage 1
0xA1	Collection	0x01	Application
0x19	Usage Minimum	16	
0x29	Usage Maximum	16	
0x15	Logical Minimum	0x00	
0x25	Logical Maximum	0xFF	
0x75	Report Size	8	field bits
0x95	Report Count	16	
0x81	Input	0x00	Data, Array, Abs
0x19	Usage Minimum	16	
0x29	Usage Maximum	16	
0x95	Report Count	16	
0x91	Output	0x00	Data, Array, Abs
0xC0	End Collection		

Table 10: HID IN *Report* descriptor

```
typedef struct {
  unsigned long   timestamp;     // milliseconds
  struct {
    signed short    dx;          // X motion
    signed short    dy;          // Y motion
  }               sensorMotion; // sensor motion
  struct {
    unsigned short  x;           // X position
    unsigned short  y;           // Y position
  }               sensorPos;     // sensor position
  signed short    incencMotion; // encoder motion
  unsigned short  incencPos;     // encoder position
} APP_HID_TX_REPORT;
```

Figure 17: HID IN report semantics

LED module The LED module simply drives the three LEDs on and off.

Encoder module A small software module (*incenc*) decodes the quadrature signal fed by a rotary encoder, or exploited from a properly timecoded media stream.
The quadrature encoder waves are gathered by the MCU by triggering interrupts in a rather tricky way. The two quadrature inputs, namely *A* and *B*, are connected to the negative inputs of the two comparators of the chosen MCU. The comparators have an internal reference (positive input) voltage fed by the internal DAC module at $V_{DD}/2$, with a small hysteresis.
Whenever the quadrature signal (A or B) crosses the reference voltage, an high priority interrupt is generated. The ISR detects the current quadrature phase and accumulates the single step into a global delta counter.
The delta counter can then be collected by the (slower) HID report generator, which resets it.
The initialization routine simply configures the voltage reference DAC, the two comparators, and interrupts.
There is no background service, since all the processing is triggered by interrupt events.

Sensor module Another software module (*adns2080*) handles the mouse sensor. This module provides communication rountines over the SPI port of the MCU in a half-duplex fashion.
All the communication routines are blocking, but this is not a problem because they are executed in the background service (main loop) of the firmware architecture.
An initialization routine configures the SPI port, then the mouse sensor. Its setup will allow for top-performance (no low-power states), 12-bits deltas reporting, and active-low level-sensitive motion interrupt generation. This routine will also check for proper communication with the daughter board.
The high-priority motion interrupt is generated by the sensor whenever motion is detected. This interrupt is cached by raising a firmware flag. The high priority level will minimize CPU cycles for this very simply operation.
The background service polls for the interrupt flag, and starts a *motion burst* read, which collects motion deltas over the serial port by minimizing dead times. Deltas are accumulated on global counters, which will be gathered by the HID report generator and thuse reset.

Main module The main module, also called the *app*, initializes the system, handles the main loop, and provides some interrupt functions.
The initialization sequence calls the initialization of all the sub-modules.
The main loop cycles through the *service task* of the sub-modules, so that motion deltas are computed for both the sensor and incremental encoder modules. If the deltas are meaningful and the HID transmission endpoint is available, a new HID report is built and sent to the host.
The two interrupt handlers, one for high-priority and one for low-priority interrupts, will obviously handle events generated by peripherals.
Some functions are dedicated to locking and unlocking of the application resources, and are basically wrappers respectively to global interrupt disable and enable.

Tasks organization The modules are organized in a *while-loop* fashion, because the chosen MCU has not enough computational power, nor a stack manageable by an actual RTOS.
An initialization procedure will turn all the modules on, and enables interrupts. After that, the main loop is entered. Inside the main loop, incremental encoder and mouse sensor deltas are gathered and, if the USB endpoint is free, sent by an HID report.
Fast and simple interrupt events are processed by the high-priority ISR, while communication events, which are slower, are processed by the low-priority ISR.

5.3 Software architecture

Thanks to the adoption of the HID standard, host software can communicate with the device easily. In fact, all the major operating systems have full support for the USB stack, including the HID application protocol.
The target software for RATT is a *vinyl emulation software*. There is a huge load of programs to emulate traditional DJ consoles, but not so many support HID controllers, and fewer let the user define his own mappings. Among all of them, the reference software is *Virtual DJ* by *Atomix Productions* [20], available for both Microsoft Windows and Apple MacOSX.

HID/USB connectivity Connection between Virtual DJ and RATT is straightforward. This software has native support for HID devices, which are *plug and play* and *hot-pluggable*.
After connection, Virtual DJ searches for a valid mapping for the device. If found, it is immediately loaded, otherwise the invalid device is simply ignored.

Device description In order to be considered, a device must supply the device definition to Virtual DJ, in this case for a HID device [21] as seen in **Figure 18**. The VID and PID are the actual HID device identifier, and the HID report size is specified. The device supports only one deck at a time.
The device description is divided in four pages. The *init* and *exit* pages should contain information for both initialization and deinitialization respectively, and are ignored. The *out* page is ignored too, because the device does not receive any meaningful messages from the host software.
Instead, the *in* page contains the description of the three motion sources. They are all interpreted as jogwheels, with full 16-bit unsigned counters which keep track of the accumulated position. The *full* attribute tells the jogwheel CPR, and needs to be calibrated with geometrical parameters of the wheel itself.
On Windows, the XML text must be saved in:
`%UserProfile%/Documents/VirtualDJ/Devices/`
while on MacOSX in:
`~/Documents/VirtualDJ/Devices/`
for example as `ratt_device.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<device name="RATT" author="Andrea Zoppi" decks="1"
 type="HID" vid="0xDEAD" pid="0xBEEF"
 reportsize="16">
  <page type="init">
    <!-- Nothing to initialize -->
  </page>
  <page type="in">
    <!-- Optical motion sensor X position -->
    <jog name="SENSOR_POS_X"
     byte="8" size="word" endian="little"
     full="24000" min="0" max="65535"/>
    <!-- Optical motion sensor Y position -->
    <jog name="SENSOR_POS_Y"
     byte="10" size="word" endian="little"
     full="24000" min="0" max="65535"/>
    <!-- Incremental encoder position -->
    <jog name="INCENC_POS"
     byte="14" size="word" endian="little"
     full="4000" min="0" max="65535"/>
  </page>
  <page type="out">
    <!-- No outputs -->
  </page>
  <page type="exit">
    <!-- Nothing to deinitialize -->
  </page>
</device>
```

Figure 18: Virtual DJ device definition for RATT

Device mapping Once the device is recognized by Virtual DJ, it needs to be *mapped* [22]. The mapping process associates a control event, declared by the device definition, to some actions. These actions are executed by parsing the *VDJscript* syntax [23].
For demonstration purposes, only the optical motion sensor X position will be processed as jogwheel position by the software, while the Y position and the quadrature position will be ignored.
The XML text in **Figure 19** summarizes the actions being mapped. On Windows, it must be saved in:
`%UserProfile%/Documents/VirtualDJ/Mappers/`
while on MacOSX in:
`~/Documents/VirtualDJ/Mappers/`
for example as `ratt_mapping.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<mapper device="RATT" author="Andrea Zoppi"
 description="RATT" version="704" date="10/08/2012">
  <!-- Use the sensor X position as a jogwheel -->
  <map value="SENSOR_POS_X" action="jogwheel"/>
  <!-- Ignore the sensor Y position -->
  <map value="SENSOR_POS_Y" action="nothing"/>
  <!-- Ignore the incremental encoder input -->
  <map value="INCENC_POS" action="nothing"/>
</mapper>
```

Figure 19: Virtual DJ device mapping for RATT

5.4 Sizing and computations

The device must be able to keep track of turntable movements. This can be achieved only if the platter is moving

at less than the maximum speed that the optical motion sensor can reach. Given this constraint, it is possible to determine the radius at which the center of the sensor can be placed. In the following, some computations will lead to the expected results.

Turntable Nominal turntable data can be found in **Table 11**. Nominal settings are for a 12.00 inches disc, spinning at 33.00 rotations per minute, with no pitch correction.
Computations are done with well-known basic equations.

Name	Description	Value	
ttDiscDiam	Disc diameter	12.00	in
ttNomAngSpeed	Nominal angular speed	33.00	rot/min
ttPitchOff	Pitch offset	0.00	%
ttDiscPerim	Disc perimeter	37.70	in
ttAngSpeed	Angular speed	33.00	rot/min
ttRevRate	Revolution rate	0.550	Hz
ttRevTime	Revolution time	1.818	s
ttTgSpeed	Tangential speed	20.73	in/s

Table 11: Nominal turntable data

Sensor Optical motion sensor computations are very easy. Given the sensor resolution and maximum speed, it is possible to know the maximum theoretical dots count per second.
Also, the HID configuration of the device indicates a polling interval of 1 ms, so it is useful to know the maximum counter value in that that time interval. Since RATT is using a 12-bits report count, counters will never be saturated even with some jitter in the main loop of the firmware.
Data can be found in **Table 12**.

Name	Description	Value	
ssRes	Resolution	2000	dot/in
ssMaxSpeed	Maximum speed	30	in/s
ssMaxDotRate	Maximum dots rate	60000	dot/s
ssMaxDotOneMs	Maximum dots in 1 ms	60	dot

Table 12: Optical motion sensor data

Measurement Finally, it is possible to define the measurement environment. The center of the motion sensor must be placed at an appropriate distance from the disc center, so that motion can always be detected correctly. This means that the speed of the disc below the sensor is always lower than the maximum speed detectable by the sensor itself.
Consider the case of the Y direction of the sensor being perpendicular to the tangential velocity. The X counter represents the number of virtual dots been measured by the sensor until that time, while the Y counter should always be clear.
The disc is supposed to spin at nominal angular speed, so after a single full revolution the X counter tells how many dots make the perimeter at that radius.

By choosing a measurement radius of 4.00 inches, even when spinning at twice the nominal speed the sensor should measure the speed correctly. In fact, the speed is less than half of the maximum reachable by the sensor.
Results are shown in **Table 13**, while an example of placement can be seen in **Figure 20**.

Name	Description	Value	
mmRadius	Radius	4.00	in
mmAngleOff	Angle offset	0.00	°
mmRadiusCm	Radius [cm]	10.16	cm
mmSpeed	Speed	13.82	in/s
mmDotRate	Dots rate	27646	dot/s
mmRevDots	Revolution dots	50265	dot

Table 13: Nominal measurement data

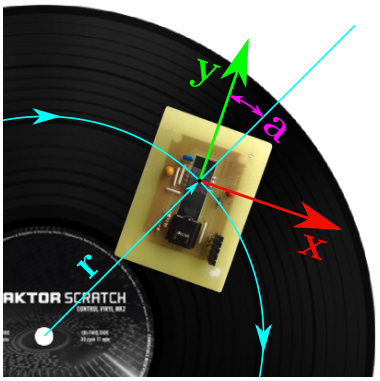

Figure 20: Sensor placement over a 12” vinyl record

5.5 Field results

Some empirical test were performed to check the goodness of the chosen approach.
First, some timing tests will determine if the device is rapid enough to handle fast changes in speed, without delays noticeable by an average DJ.
Moreover, some space tests will show if the motion sensor alone is capable of keeping track of absolute motions, with a small accumulated displacement error.
Finally, an opinion about the overall feeling of the device behavior will be given.

Motion detection rate A first test measures the interval between *MOTION* interrupts called by the optical motion sensor. The sensor does not have a fixed acquisition clock, but instead it is adapted on the speed of the moving surface beneath.
When moving at a fairly slow speed, the interval between interrupts is around 3.7 ms (4 ms inside the datasheet) as seen in **Figure 21**.
When the moving surface gets faster, the interrupt generation slows down at about half that speed, with an interval around 7.2 ms, as shown in **Figure 22**.

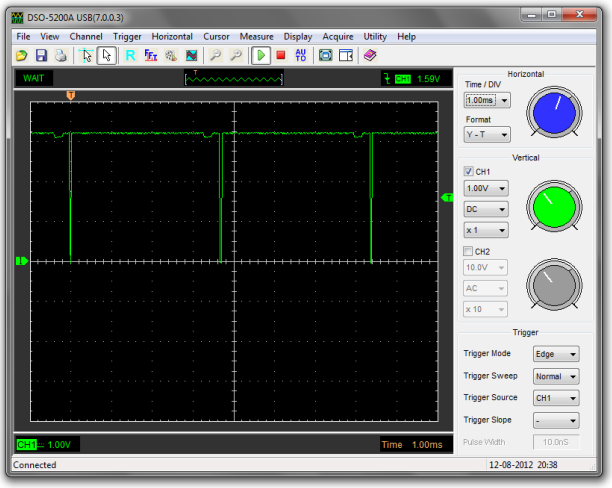

Figure 21: *MOTION* interrupt delay for slow steady speed

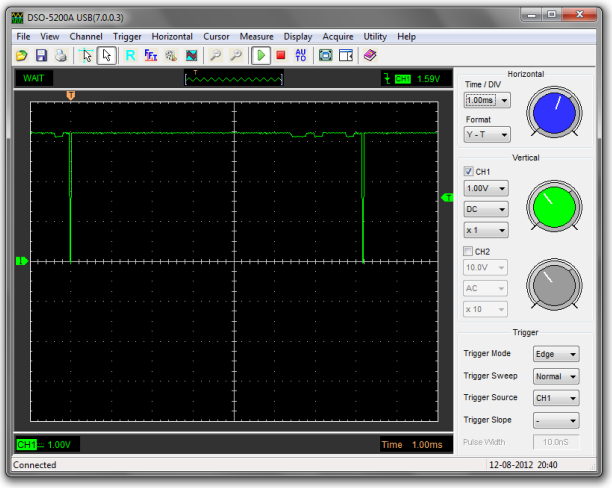

Figure 22: *MOTION* interrupt delay for fast steady speed

Motion processing time After being issued, the *MOTION* interrupt has to be processed. The sensor is configured so that the interrupt signal goes low when motion is detected, then is kept low until all the motion counters are read, going back high after the reading.

By measuring this delay, it is possible to know how long it takes for the MCU to read those counters after the *MOTION* interrupt is reported. As captured in **Figure 23**, the processing takes typically 30 µs, with a maximum of 72 µs in **Figure 24**.

This delay is clearly negligible, and does not affect the overall latency nor the internal motion sensor counters.

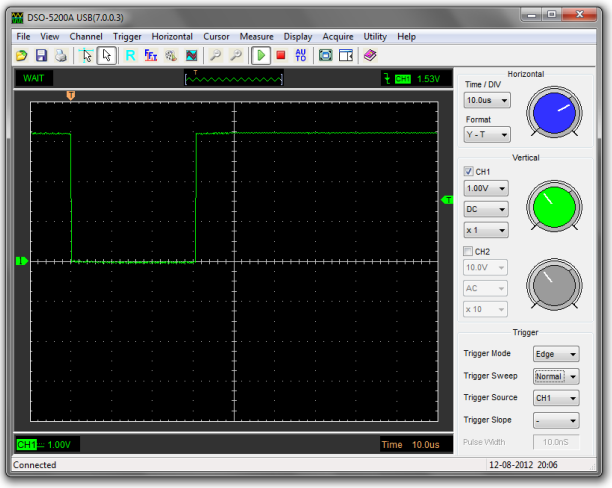

Figure 23: Typical *MOTION* pulse duration

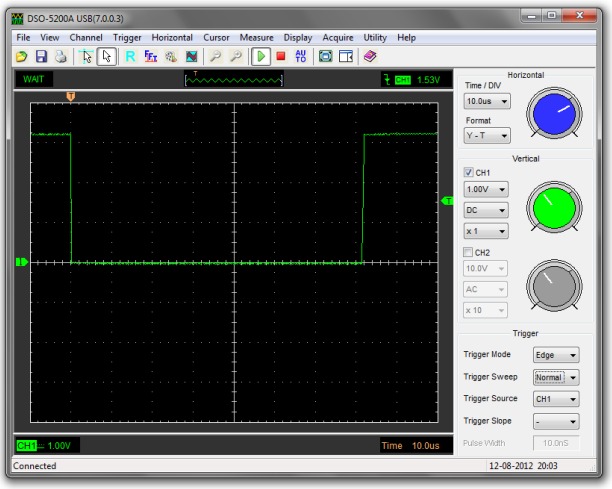

Figure 24: Maximum *MOTION* pulse duration

HID motion processing time A final timing test will prove that the MCU is fast enough to handle HID transactions in time.

As an extension of the previous test, the measured delay adds the time needed to build and buffer the HID report into the hardware USB RAM. A LED signal was used to draw the processing length.

The minimum and maximum delays can be extracted by **Figure 25** and **Figure 26**. The delay is thus in the range of 74 µs to 110 µs, which is still negligible for the purpose.

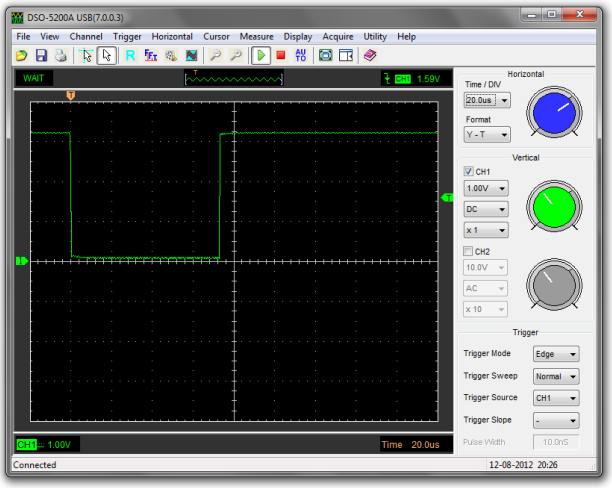

Figure 25: Minimum *MOTION* interrupt total time

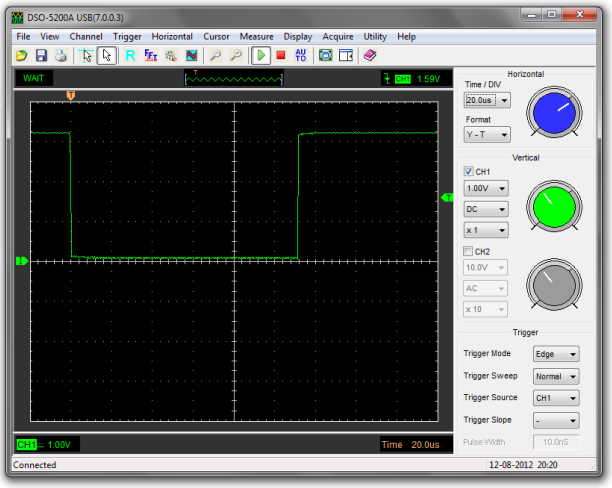

Figure 26: Maximum *MOTION* interrupt total time

Absolute tracking **TODO…** absolute position after 10 scratches

Overall feeling The overall feeling is good. The latency is low enough – comparable to that of the soundcard, which is still dominant – and it is possible to scratch with ease, even though very fast scratches feel a bit odd.

Anyway, when trying to reposition the virtual needle the precision is very high, altough the absolute tracking of the sensor alone is not as accurate as with the help of an incremental encoder. Such a good precision is comes handy also when using the jogwheel to push/pull the virtual vinyl – usually called *nudge* operations.

The software seems to handle the workload with no stress on the CPU, even with high deltas.

6 Advanced proposed approach

Due to the few capabilities of the simplified approach, which is just for basic research and demonstration purposes, a more advanced way to achieve better results is presented in the following. No prototype was made, but a high-level description of the target architecture can drive the development of an actual device.

Basically, the advanced approach exploits the computational power of the most recent MCUs, in order to provide faster motion detection and processing rate, as well as the opportunity to handle tasks outside those for the motion detection.

The whole system will be capable to manage a wide range of input devices – buttons, sliders, jogwheels – and some displayed information. The workload is split into simple *tasks*, handled by a *Real-Time Operating System* so that it will be easier to develop the firmware.

6.1 Hardware architecture

The generic architecture reflects that of complex digital DJ controllers which offer jogwheels (often touch-sensitive), but also buttons, knobs, sliders, lights, displays, external connections, audio piping to DSP, and so on.

By choosing a fully-featured MCU of nowadays, such as the *STM32 F4* series [24], it is possible to handle almost all these devices with only one MCU, at reduced overall price. Obviously, some additional chips are still needed – motion sensors *in primis* – but the whole architecture can be shrunk into a few chips.

Digital inputs The most basic type of inputs is *discrete-state (digital)* input devices, such as buttons, switches, toggles, and so on. In order to handle them, a *matrix* topology often suffices, such as the one seen in **Figure 27**. Such matrix can be scanned row-by-row, active low logic, with full support for multiple elements active at the same time.

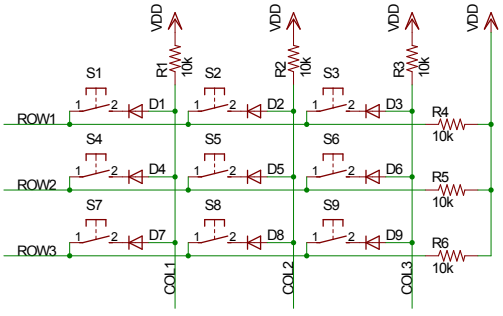

Figure 27: 3×3 keypad, active low, multiple key presses

Analog inputs The user should also be able to control DJ software parameters with a continuous values range. Such values are provided through knobs and sliders. Electrically, these devices are all potentiometers connected to

the ADC module of the MCU through its analog multiplexer, as shown in **Figure 28**. The analog values will then be converted by the ADC and processed by the *analog inputs task* of the firmware.

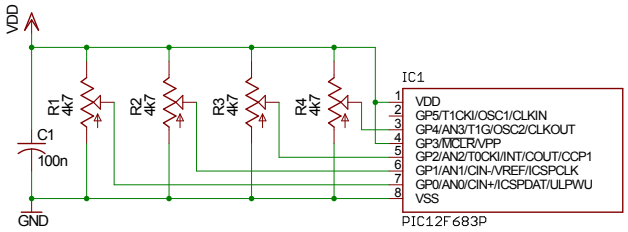

Figure 28: 4 potentiometers connected to a MCU, through its built-in analog multiplexer and ADC

Jogwheels Jogwheels are fundamentally those developed with the simplified approach at Section 5. Each jogwheel is made by a medium-resolution optical quadrature wheel and its encoder, plus an optical motion sensor.
The quadrature encoder, thanks to its fairly low speed, can be handled by MCU interrupt signals (edge-triggered, both signs).
Instead, motion sensors must be connected to the appropriate digital bus (SSP, SPI), and additional pins to suitable inputs.
Jogwheels should also provide touch-sensitivity, so that a digital signal (see above) can be generated when the user puts fingers on the wheel. There are different ways to provide touch sensitivity, but will not be covered in this document.
In order to shrink the design for low-end controllers, it is possible to choose a smaller sensor package. For example, the *ADNS-3550* by *Avago* [25] (see **Figure 29**) features an *ultra slim* design. This may reduce manufacturing costs, because it integrates the LED, has a very small size, and is suitable for *SMD* soldering. As a drawback, it is addressed to ultra-low-power wireless devices, so its performance is worse than the average sensor used for the prototype – up to 20 in/s at 1000 DPI instead of 30 in/s at 2000 DPI – but is still very good for medium-sized jogwheels.

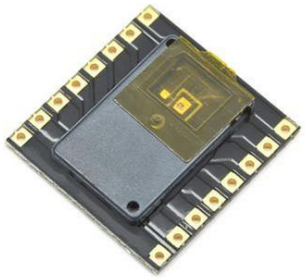

Figure 29: Avago ADNS-3550, an *ultra slim* motion sensor

Lights Some light can help the user in keeping track of some DJ software states. Thanks to their very low current consumption and simplicity, LEDs are always the best choice.
Simple lights can be driven directly by the MCU digital pins, in any desirable fashion – could it be direct coupling, LED matrix, or charlieplexing. These lights can be either on or off.
Dimmed lights are more complex to handle, and need to be connected to PWM outputs to give an intensity effect to the human eye, proportional to the PWM duty cycle. Due to scarcity of such outputs, this feature is often not implemented at all, or applied to simple behaviors – for example, the *M-Audio Torq Xponent* [14] gives the so-called *Christmas tree* effect, with all LEDs beign modulated by the music tempo.

Displays Very advanced controllers sometimes provide information through one or more displays. There are so many display types on the market, that it is difficult to suggest one. Anyway, for pure text displays, or generally soft-realtime information visualization, the connection to one among the common UART, I2C, SPI buses is enough.

6.2 Firmware architecture

The firmware architecture relies on those capabilities given by an RTOS. With such an operating system, it is possible to divide the firmware into isolated entities, which collaborate to fulfill the final goal.
Depending on the device being engineered, there is no optimal firmware architecture, but a general one will be described.

RTOS overview A RTOS-based firmware can be split into the following entities: tasks, drivers, and concurrent data structures.
A *task* is a routine which runs continuously. It can receive, process, and send data to other entities (tasks or drivers). More tasks can run at the same time. If it is not possible to run more tasks contemporaneously, the RTOS must be able to approximate this behavior by an appropriate *scheduling*. Also, in some scheduling policies a taks is given a *priority*, so that those with a higher priority can suspend those with a lower one, and take control of the required resources. Suspendable tasks are called *preemptive*.
A *driver* is a collection of routines which control peripherals, and give a standardized abstract interface to the user. A driver can rely on both tasks and *interrupt service routines*, being seen as part of the OS – instead, the user code is always based on tasks, because ISRs are always processed by drivers.
Concurrent data structures are used to exchange data among tasks and drivers. Because of concurrency, tasks and drivers must be able to exchange data without corruption, through *atomic* operations. In order to achieve atomicity, a *locking system* is implemented by such data structures, handled by the RTOS. These data structures can span

from simple semaphores to complex data queues.
A very important feature of a RTOS is its ability to *schedule* tasks deterministically. Tasks are often assigned a fixed period, which should expire with the lowest uncertainity. By choosing a RTOS scheduling policy, and with a compatible workload, such predictability can be achieved.
A global logical view of the firmware architecture is depicted in **Figure 30**.

Digital inputs Digital inputs are very easy to handle. Supposing a matrix hardware topology, the *Digital task* simply cycles through all the rows and colums to detect changes, operations done by the *Digital driver*.
Since in audio applications timing is critical, this loop has to be fast enough so that the user will not notice delays. The Digital task is pretty simple, so it can be given high priority, and the cycle period will match that of HID packets.
The state of digital inputs is held by a double-buffered record, which is then read by the Controller task.

Analog inputs Analog inputs require an *ADC driver* to handle the ADC and its analog multiplexer.
The *Analog task* polls each analog input for changes. Since human hand movements have a dynamic rate lower than 10 Hz, a sampling rate of 200 Hz (5 ms) per channel is enough to keep track of smooth motions. Further smoothing can be performed with some digital filtering, but is not required.
Like for digital inputs, the state of analog inputs is kept by a double-buffered record, read by the Controller task.

Lights module A module managing lights can be added to the fimware architecture.
A driver will provide a suitable software interface to lights. It might support simple on/off lights, but also modulated lights. In the case of lights modulated by PWM outputs, it might lean on the PWM driver of the RTOS.
In the case of charlieplexing, but also for modulation, there is a need to refresh periodically some hardware peripherals inside a task, to have light correctly lit. This task could be given a low priority, because delaying the deadline for a few milliseconds is not that critical. Anyway, the refresh rate should be high enough to give a smooth transition between light states, around 60 Hz ($\approx$ 16 ms) for the best feeling and user reaction time.
The Controller task tells the state of lights to the Lights task through a double-buffered record.

Encoder module As for the simplified approach, when requiring absolute jogwheel positioning it is possible to add support for incremental encoders. Such encoders output quadrature waves, which can be used to increment or decrement some firmware counters. Again, edge-triggered interrupts should be used to manage counters.
The *Encoder driver* will handle such interrupts, and it will give access to counters atomically.
The *Encoder task* awaits for interrupt signals by the driver, occurring when the quadrature wave performs a transition. The internal counter accumulates the step, so that the absolute position is tracked. This task is the most critical, because no steps can be lost, and is given the highest priority with instant response.
The Encoder task tells the current position and deltas to the Controller task by the use of a double-buffered record.
There can be multiple incremental encoders in the design.

Sensor module The sensor module will keep track of the position estimated by the optical motion sensor.
The *Sensor driver* will communicate to the sensor through the SSP bus, which in turn is managed by the *SSP driver*.
The *Sensor task* can have different behaviours, depending on the capabilities of the sensor. If it features a motion interrupt, like the one chosen for the simplified approach, the sensor awaits for it, then retrieves the motion delta. If such an interrupt is not present, the Sensor task has to poll continuously for new data at a very fast rate. This case is typical for pro-grade sensors, which have a very high framerate, so it is most convenient to run a tight polling loop at high speed.
The Sensor task tells the current position and deltas to the Controller task by the use of a double-buffered status record.
There can be multiple optical sensors in the design.

HID/USB module In order to communicate with the host the host, HID reports are processed by the *HID driver*, which in turn is built on top of the *USB driver*. Such drivers will simplify the packet management by the user, also by providing a template *HID task*. This task will exchange HID reports with the Controller task, in the form of plain data records. Such record will be marshalled and sent to the USB host through the aforementioned drivers.
HID processing should be faster than the report generation of the Controller task, so the HID task has a higher priority. Also, reaction time should be minimized.

Console module A console module could be added for debug purposes only. By the way, an advanced hardware debugger is almost always present in MCUs today, so there is no real need for a debug console inside a consumer DJ controller.
If the console module is added, it needs to be interfaced to the UART through a driver provided by the RTOS. It also needs a task with low priority in the case of a simple debug console – if necessary, immediate commands can always be issued through the hardware debugger. Response time to incoming messages should be the shortest possible, as well as outgoing messages should be sent immediately.
The *Console task* manages console commands and interactions with the *UART driver*. Messages, in the form of

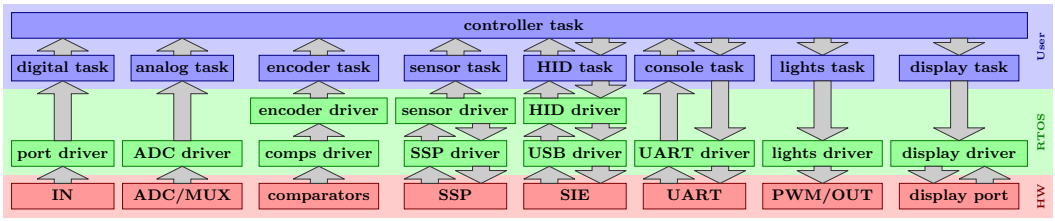

Figure 30: Logical view of the advanced firmware framework

user-friendly data structures, are exchanged between the Controller and the Console tasks. The latter performs marshalling with the UART streams.

Main module The main module is composed by the *Controller task*. It collects the status of user inputs to generate a HID report, handles incoming commands, sends data to be displayed to screens and lights, and so on. It is the core of the whole device, which performs all the data processing among the other modules.

Data collection and HID report generation should occur at most each 1 ms, so that the latency is minimized to that of USB *Full Speed* interrupt polling.

The Controller task also manages commands of the Console and Debug tasks, which must be processed in a very short time – this is why bare raw data structures are used instead of strings to be interpreted.

Tasks summary The **Table 14** summarizes the meaningful settings to be assigned to tasks inside the chosen RTOS, as described by the previous paragraphs.

Of each task, its *weight* is extimaed, which is a coarse approximation of the complexity of operations it should perform. It is not a RTOS setting, but is useful to identify the workload of tasks.

Moreover, the *priority* is intended for a preemptive and priority-based scheduler, which is common for a state-of-the-art RTOS.

At last, the expected *response time* indicates the target period of the task routine. For routines heavily based on semaphores – typically activated by interrupt signals or software messages – no delay is expected, so it is marked as *immediate*.

Task	Weight	Priority	Response time
Encoder	1	5	immediate
Digital	2	4	1 ms
HID/USB	5	4	immediate
Controller	4	3	1 ms
Sensor	3	3	immediate
Analog	3	2	5 ms
Console	4	1	immediate
Lights	2	1	16 ms
Display	5	1	16 ms

Table 14: RTOS settings of tasks

7 Conclusions

A simplified approach supporting a single jogwheel was developed. Its performance is suitable for the average DJ, achieving good tracking even with cheap COTS. Initial motion latency is fairly high, due to the power-saving states entered by a mouse sensor for wireless appliances, but is still accetpable. Motions are tracked with outstanding precision and accuracy. Repeatability is on the average, approximating absolute positioning with some errors, but still acceptable. An optional incremental encoder can help the tracking of absolute movements, through some software data processing. Software can be interfaced to the device with ease, thanks to the adoption of the HID protocol over the USB bus.

Furthermore, an advanced approach was proposed. It will support not only the kind of jogwheel presented in the simplified approach, but also peripherals of various nature. The advanced device needs a more powerful MCU to handle all those peripherals. In order to make firmware programming more flexible and easier to manage, a RTOS should be chosen. Thanks to this advanced approach, it is possible to develope a whole commercial DJ controller, with cheap yet accurate jogwheels.

References

[1] Smithson Martín: Emulator.
http://www.smithsonmartin.com/products/emulator/

[2] hexler.net: TouchOSC.
http://hexler.net/software/touchosc

[3] Stanton: SCS.3D.
http://www.stantondj.com/stanton-controllers-systems/scs3d.html

[4] Denon: SC-3900 digital media turntable and DJ controller.
http://www.dm-pro.eu/en/denondj/products/86/sc3900-digital-media-turntable-dj-controller/

[5] Numark: V7 motorized turntable.
http://www.numark.com/product/v7

[6] Eks: Otus Plus.
http://eks.fi/product.php?p=products&id=64

[7] Pioneer: DDJ-S1.
http://www.pioneer.eu/it/products/44/216/34209/DDJ-S1/index.html

[8] Vestax: VCI-400.
http://www.vestax.com/v/products/detail.php?cate_id=186

[9] Hercules: DJ Control MP3 e2.
http://www.hercules.com/uk/DJ-Music/bdd/p/110/dj-control-mp3-e2/

[10] Rane: Serato Scratch Live.
http://serato.com/scratchlive/

[11] Native Instruments: Traktor Scratch Pro 2.
http://www.native-instruments.com/#/en/products/dj/traktor-scratch-a10/

[12] MIDI: Messages.
http://www.midi.org/techspecs/midimessages.php

[13] USB: documentation.
http://www.usb.org/developers/docs/

[14] M-Audio: Torq Xponent.
http://www.m-audio.com/products/en_us/TorqXponent.html

[15] USB: HID Information.
http://www.usb.org/developers/hidpage

[16] USB: Device Class Definition for Human Interface Devices (HID).
http://www.usb.org/developers/devclass_docs/HID1_11.pdf

[17] Microchip: PIC18LF14K50.
http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en533924

[18] Avago Technologies: ADNS-2080.
http://www.avagotech.com/pages/en/navigation_interface_devices/navigation_sensors/led-based_sensors/adns-2080/

[19] Microchip: Microchip Application Libraries.
http://www.microchip.com/mal

[20] Atomix Productions: Virtual DJ.
http://www.virtualdj.com/products/virtualdj/index.html

[21] Atomix Productions: Virtual DJ Controller Definition for HID.
http://www.virtualdj.com/wiki/ControllerDefinitionHID.html

[22] Atomix Productions: Virtual DJ Controller Mapping.
http://www.virtualdj.com/wiki/ControllerDefinition.html

[23] Atomix Productions: VDJscript.
http://www.virtualdj.com/wiki/VDJscript.html

[24] STMicroelectronics: STM32 F4.
http://www.st.com/stm32f4

[25] Avago Technologies: ADNS-3550.
http://www.avagotech.com/pages/en/navigation_interface_devices/navigation_sensors/led-based_sensors/adns-3550/