RATT - Relatiely Accurate TurnTable

Relative turntable motion detection with 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 when detecting very fast rotations, because they become 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 grew considerably. By the way, cheap digtital 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 [**?**] [**?**] or capacitive [**?**] controllers, which follow the market wave of touchscreen devices. There are also some evolutions of the optical jogwheels, which are motorized [**?**] [**?**] 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 [**?**] [**?**] [**?**].
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.

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*[1] these inputs to some software parameters, 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, **TODO...** 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 **[CITEME]**).

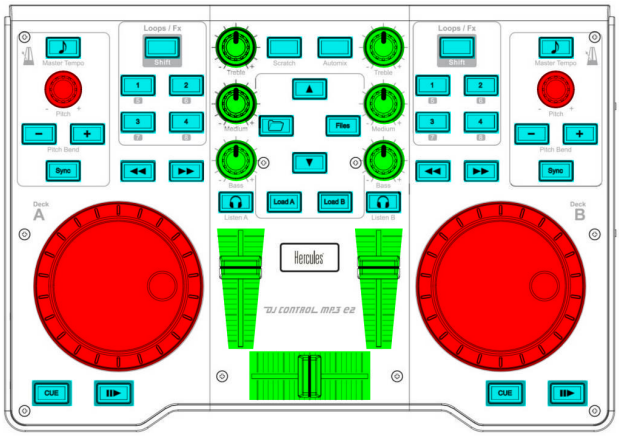

Figure 1: *Hercules DJ Control MP3 e2*[**?**] intereface schematic with buttons (cyan), incremental encoders (red), sliders and knobs (lime)

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-detector sensors mounted on the chassis. The light-detector sensor 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.

[1] Assign a triggered action to an input event

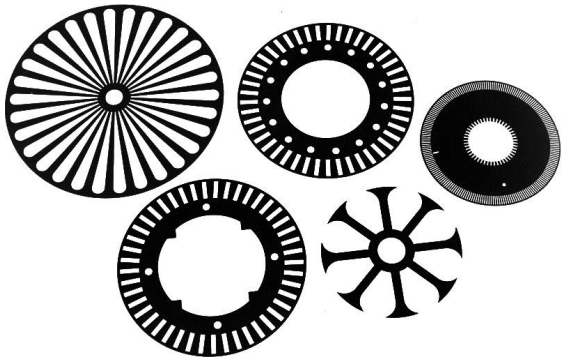

Figure 2: A set of incremental (quadrature) rotary encoder wheels

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 sensors 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**.

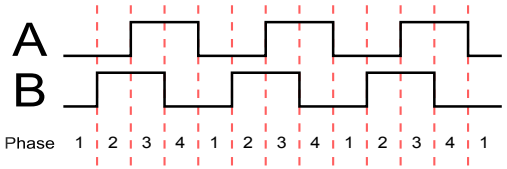

Figure 3: Quadrature pattern, going forward left-to-right

An example of jogwheel internals can be seen in **Figure 4**. It is a close-up of the *Vestax VCI-400*[**?**] 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-detector sensors 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 (described further). 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 [**?**][**?**] 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 but 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.

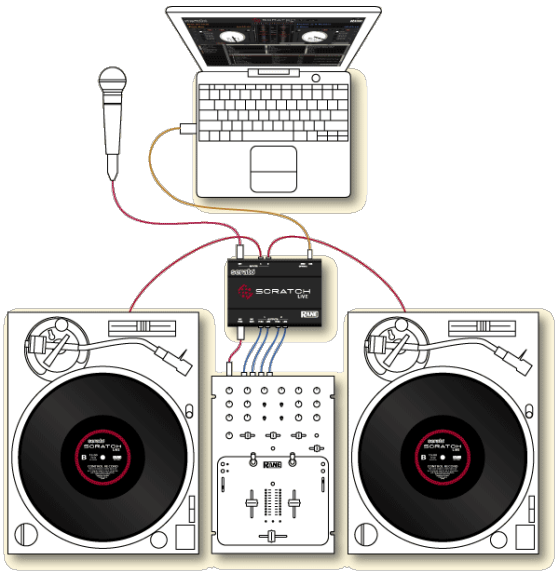

Figure 5: *Serato Scratch Live* setup with two turntables

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*[**?**] timecode signal (aka *noisemap*), with some amplitude-modulated bits at the sine wave top peaks.

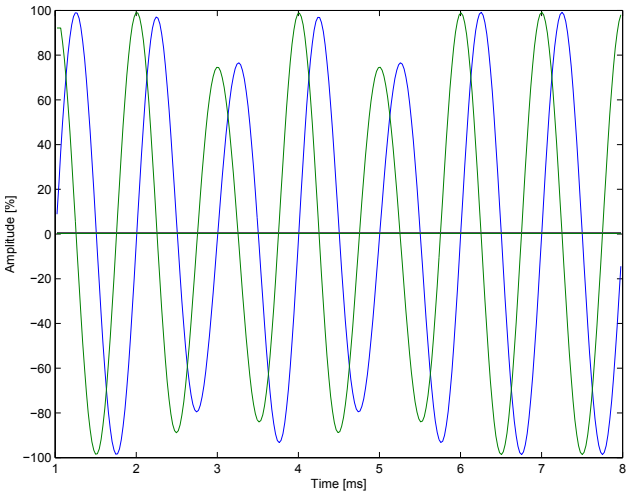

Figure 6: Slice of the *Serato Scratch Live* timecode

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 whol 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.

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 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 pont where to start playing from.
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. Unfortunately, the performance was not that good, mainly because of latencies in both the MCU and the sensor. This is why an advanced approach will be proposed in the next section, which should provide a much better response and could handle more peripherals then the only wheel.

3.1 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 timecode 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 hosts the MCU, a *PIC18LF14K50***[CITEME]**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 provided 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.
Finally, to communicate with the sensor module some control and interrupt signals, and a half-duplex serial port are provided.
Figure PLACEHOLDER

Sensor board The sensor board simply hosts the mouse sensor and the surface illumination LED. The sensor is an *ADNS-2080***[CITEME]**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 maximum latency because the clock speed is variable, but it can be forced to maximum performance through communication.
Figure PLACEHOLDER

Timecode preamp board An optional board was designed to properly amplify a *Line* – no *Phono*! – timecode signal so that it can be properly 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 PLACEHOLDER

3.2 Firmware architecture

Due to the lack of a real *Real-Time Operating System* (*RTOS*) on the chosen MCU, the application was written in a *while-loop* fashion, with *interrupt-driven* tasks at *high priority*.

Bootloader In order to simplify firmware deployment, a HID bootloader was added to the development prototype. This will occupy the first 2048 words of the program space, which is not that small. The bootloader is the one provided by the *Microchip Application Libraries* [**?**], 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 TODO...

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.

Remarks **TODO…**

3.3 Software architecture

TODO…

HID/USB connectivity **TODO…**

Device description **TODO…**

Device mapping **TODO…**

3.4 Sizing and computations

TODO…

3.5 Field results

TODO…

4 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.
TODO…

4.1 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* family, 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 *button matrix* often suffices, such as the one seen in **Figure 7**. Such matrix can be scanned row-by-row through row (de)multiplexing, and then processed by the *digital inputs task* of the firmware **[CITEME]**.

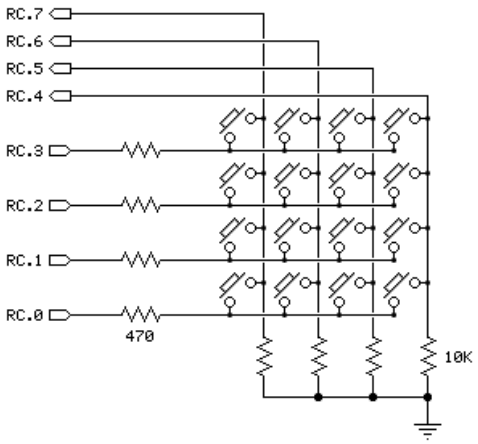

Figure 7: A 4×4 matrix keypad driven by an 8-bits port

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 [CITEME]**. The analog values will then be converted by the ADC and processed by the *analog inputs task* of the firmware **[CITEME]**.
Figure PLACEHOLDERanalog inputs

Jogwheels Jogwheels are fundamentally those developed in the simplified approach at Section 3. 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 (I2C, 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.

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*[**?**] 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.
TODO…

4.2 Firmware architecture

TODO…

Main module **TODO…**

HID/USB module **TODO…**

LED module **TODO…**

Encoder module **TODO…**

Sensor module **TODO…**

Debug module **TODO…**

Tasks organization **TODO…**

Remarks **TODO…**

5 Appendix A - Common MCUs brief survey

TODO…

5.1 Microchip PIC18

TODO…

Common features **TODO…**

Pros **TODO…**

Cons **TODO…**

5.2 Microchip PIC24

TODO…

Common features **TODO…**

Pros **TODO…**

Cons **TODO…**

5.3 Microchip PIC32

TODO…

Common features **TODO…**

Pros **TODO…**

Cons **TODO…**

5.4 Atmel AVR

TODO…

Common features **TODO…**

Pros **TODO…**

Cons **TODO…**

5.5 Atmel AVR32

TODO…

Common features **TODO…**

Pros **TODO…**

Cons **TODO…**

5.6 TI MSP-430

TODO…

Common features **TODO…**

Pros **TODO…**

Cons **TODO…**

5.7 ARM Cortex Mx

TODO…

Common features **TODO…**

Pros **TODO…**

Cons **TODO…**

6 Appendix B - Optical motion (mouse) 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.

6.1 Architecture and operation

A generic mouse sensor is composed by components of various nature, as seen in **Figure 8**.

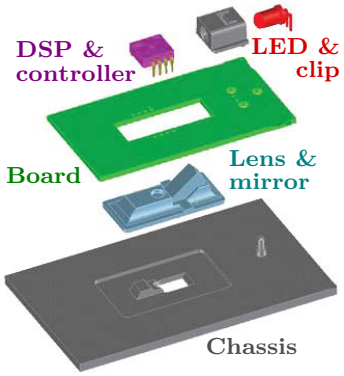

Figure 8: Common motion sensor assembly

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.

6.2 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.

I2C When requiring some more flexibility, an I2C bus is already enough for average sensors. Due to its message-based nature, it is possible not only to read internal counters, but also to get and set come configuration parameters. On the other hand, the I2C protocol introduces some unnecessary latencies due to handshaking and baud rate – standard I2C is only at 100 kb/s, fast mode at 400 kb/s.
Figure PLACEHOLDER

SSP/SPI When requiring slightly higher speed and lower latency, it is possible to adopt *Synchronous Serial Port* / *Serial Peripheral Interface* (respectively, half-duplex and full-duplex) interfaced sensors. They usually do not require handhaking, even though some control pins are often used in such a way (e.g. the common *shutdown* pin). Also, the bit rate is often higher – from 1 MHz to some tens – even though pratically not much higher than I2C.
Figure PLACEHOLDER

7 Appendix C - Controller communication protocol comparison

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

7.1 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 nowadays digital music equipment.
The hardware architecture can be seen in **[CITEME]**. Basically, it is a standard serial point-to-point connection based on common UARTs. 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 in audio timing.

Basic protocol The serial protocol is 8-bits per message word, with one start and one stop bits (low), 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 **[CITEME]**, even tough manufacturers often interpret them freely (e.g. through *remapping*).
The first word of a message (the 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. The control word identifies the message type and the channel number. The second word is usually the identifier of the control (key, knob, slider, wheel, etc.) being actioned, and the third one 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

7.2 MIDI/USB

The *Universal Serial Bus* (*USB*) **[CITEME]**is probabily the most available nowadays for consumer electronics interfacing. 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 displatching 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 **[CITEME]**, by using a single USB cable.
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 **[CITEME]**. 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 indexes 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

7.3 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*), 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.
The flexibility of HID is justified by the way its subprotocol is developed. HID messages, called *reports*, have a fixed structure, defined at the moment the HID configuration of the device is sent to the host.
TODO…

Basic protocol **TODO…**

Pros

- Highest USB priority
- State streaming
- **TODO…**

Cons

- Sill some USB delays
- Custom HID reports to parse
- **TODO…**