MIDI to RS232 adapter

The article contains two parts: a description of a circuit that translates the MIDI signal level to RS232 levels (and vice versa), and programming notes to send and receive MIDI packets. The first part, the circuit, is general purpose —although using this converter to connect a MIDI keyboard to the serial port of your PC may require a special RS232 card. The second part, though, is specific to the H0420 programmable MP3 player. In fact, the article was written specifically to connect this MP3 player to a MIDI chain.

An MP3 player in a MIDI chain? Why?

MIDI is an acronym for *Musical Instruments Digital Interface*. The standard describes the hardware interface and the protocols through which (music-producing) instruments communicate amongst each other, or with computer and other devices. The key elements for MIDI are performance and reliability in hostile environments.

The MIDI protocol carries commands and timings. It does not carry digitized audio, although a command may indicate what kind of instrument should sound on a particular channel. When you move a MIDI file from one PC or instrument set-up to another, it it is actually likely to sound entirely differently.

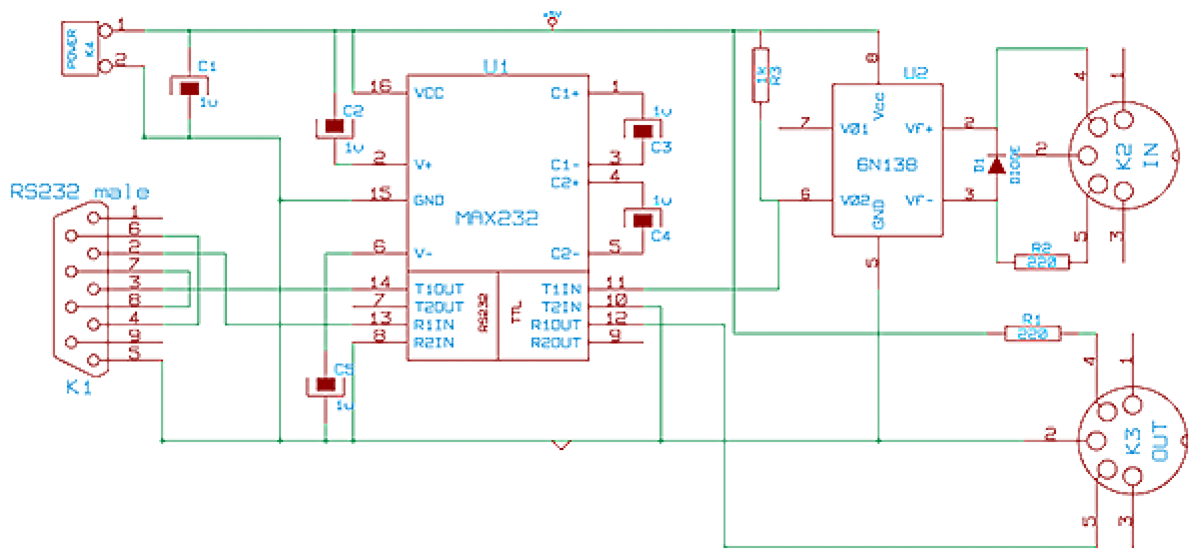
So MIDI is only loosely coupled to audio. As a communication norm, the MIDI standard is about *performance* and *timing*. The "instruments" that it links in a network are not necessarily musical instruments. Lighting colour organs and stage effects (e.g. confetti cannons) have also been linked into a MIDI network, in order to synchronize visual effects with music.

In conclusion, MIDI is a general purpose communication standard that focuses on music applications. The H0420 programmable MP3 player is a general purpose controller with a high-quality MP3 decoder & audio circuit on board. Adding this controller to a MIDI network allows you to synchronize special audio effects to other events in the music performance. In such a set-up, the H0420 receives and handles MIDI packets, but does not send them. In an alternative set-up, the H0420 monitors all kinds of other sensors that can be attached to it and sends MIDI packets to notify the entire MIDI network of these events. The H0420 can also both send and receive MIDI packets.

The circuit

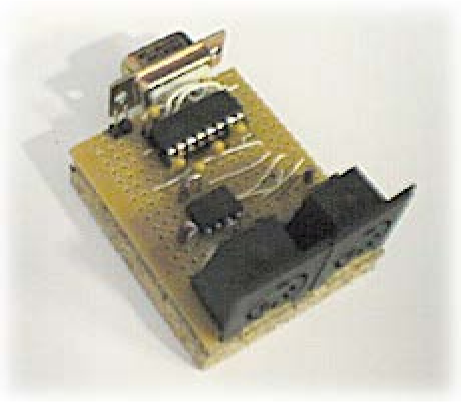
The protocol is not unlike a typical RS232 protocol: one start bit (must be 0), eight data bits, one stop bit (must be 1), no parity and no hardware handshaking. The Baud rate is 31250 bps (±1%). In order to get a fair level of immunity against electrical interference (irradiation), MIDI uses a *current loop* signal line (5mA, usually at approximately 5V), rather than the standard voltage level signals. A logic 0 is defined as current flowing, and a logic 1 as no current. This counter-intuitive definition was chosen so that an unconnected input wire does not "see" a series of start bits —a start bit is logic 0.

MIDI uses two 5-pin DIN connectors (180°) for input and output, plus sometimes a third for an unprocessed pass-through from the input. Pins 4 and 5 carry the signal, with pin 4 being the positive voltage; pin 2 is connected to the shielding of the cable on the output connector, but it is left unconnected on the input connector; pins 1 and 3 are not used. To avoid ground loops, there is an opto-isolator at the input connector.



All capacitors are 1µF. The resistors in the signal lines (R1 and R2) are 220 ohm, together with the voltage drop of the opto-coupler (6N138), these regulate the current at approximately 5 mA. The value of R3 is not critical; 1k is fine.

The circuit assumes that it will be connected to a "DCE" device. When you wish to connect it to a "DTE" device, such as a PC, you should replace the male DB9 connector (K1) by a female one and swap the pins 2 and 3 on that connector.



The circuit requires an external power source. Some alternative designs for an RS232-to-MIDI converter that I have seen extract power from (unused) handshaking or modem-control lines on the RS-232. I chose an explicit 5V connector because the H0420 MP3 player provides a 5V power source and because I feel that using signal lines as a power source is more like a hack than a design.

The closest that most serial cards and/or drivers inside PCs can come to the MIDI Baud rate is 28800 or 38400 Baud —exceeding the 1% margin that MIDI defines. Therefore PCs will typically need either a special RS232 card or an interface with a memory buffer and a Baud rate converter (typically based in a micro-controller). As an aside, the USB-to-RS232 adapters that are based on the FT232R chips from FTDI support non-standard Baud rates, but those based on the Prolific chips do not. So if you use a USB-to-RS232 converter, choose one with an FTDI chip (see the references below).

The H0420 programmable MP3 player, on the other hand, has a very flexible RS232 interface, and it supports the MIDI Baud rate directly (it also supports any other Baud rate, up to at least 115200 bps).

Sending and receiving packets

MIDI commands have a variable size with a fixed structure. Every byte carries 7 bits of information. The highest bit of a byte is *set* for the first byte in a command and clear for any follower bytes. The start of a command can therefore be detected easily. The end of a command is not marked in a specific manner; to determine weather a you have received the last byte in a stream, you can use a set of heuristics:

- Often, the size of a command can be determined by interpreting the leader byte
- When you receive another *leader byte* (a byte with the highest bit set), then, of course, preceding command had ended.
- Transferring a byte takes 10 bits on the serial line (8 data bits, 1 start bit, 1 stop bit). At 31250 bps, it takes 320 µs to send a byte. You can put a time-out at ten times this duration, and assume that a

command has ended when you do not receive more data in, say, 3 ms. To play on the safe side, you may increase the time-out value.

The last two heuristics are easy to implement on the H0420 programmable MP3 player, because the function `packetfilter()` allows you to specify the format of an acceptable packet and a time-out for waiting for (more) follower bytes. The packet specification is in a string notation that looks like a regular expression. Since MIDI defines a leader byte as one with the highest bit set, the byte value for a leader byte will be in the range 128-255. Follower bytes are, hence, in the range 0-127. A specification for the packet format is then: a byte in the range 128-255 followed by zero or more bytes in the range 0-127.

In code (with all values in hexadecimal):

```
"[`80-`ff]{`00-`7f}"
```

Sets or ranges in square brackets match a single byte; sets or ranges in braces (curly brackets) match zero or more bytes. To indicate a byte value (rather than a character), you use the *back-quote* or *grave accent* notation, with the ` character and two hexadecimal digits. You can also use the alternative pawn notations for "control characters": `` `80 `` is equivalent to `\128` (decimal) and `\x80`. However, the pawn syntaxes do not let you insert a zero byte in a string. The back-quote notation is therefore preferred.

MIDI works with channels, and a device often monitors only a single channel (plus a handful of commands that apply to all channels). A second parameter in the `packetfilter()` function allows you to specify a filter to let only those commands through that pass the filter. If you do not have such a filter, your device will see the MIDI commands of `all` devices in the MIDI network. Setting an appropriate filter lets your device see only the subset of commands that are intended for it; this simplifies the code that you have to write to interpret incoming MIDI data. It also increases overall performance, because the reject filter runs in optimized firmware code, which is bound to be faster than the interpreted script code.

Wrapping up the reception of MIDI data, there are only two things left to do: initialize the serial port and write the function that interprets the incoming packets. This latter function is called `@receivepacket()`. Below is a script that plays a track on the "note on" command and implements a "seek to position" for the "program change" command (the standard "program change" command takes two follower bytes, my modified version accepts up to four follower bytes).

```
main()
    {
    /* Open the serial port and set the Baud rate and configuration for
MIDI */
    setserial 31250, 8, 1, 0, 0

    /* set up a filter, accept only commands 9 and C and only on channel
1 */
    packetfilter "[`80-`ff]{`00-`7f}", "[`90`c0]*", 100
    }

@receivepacket(const packet[], numbytes)
    {
    if (packet[0] == 0x90)            /* 0x9 = note on */
        {
        if (packet[1] == 0)
            stop
        else
            {
            new name[20 char]
            strformat name, _, true, "track%d.mp3", packet[1]
            play name
            }
        }
    else if (packet[0] == 0xc0)       /* 0xc = program change */
        {
        new position = 0
        if (numbytes > 1)
            position += packet[1]
        if (numbytes > 2)
            position += packet[2] << 8
        if (numbytes > 3)
            position += packet[2] << 16
        if (numbytes > 4)
            position += packet[2] << 24
        seekto position
        }
    }
```

The leader byte of a command has the channel encoded in the lowest four bits. This is important for the filter expression. If you wish to receive the commands `0x9` and `0xc` on the fourth channel, the filter expression becomes `"[`93`c3]*"` (instead of `"[`90`c0]*"`). Note that MIDI numbers channels from 1 to 16, but these get encoded in the commands as the values 0 to 15.

Sending commands is quite simple: construct the packet and send the bytes individually with the function `sendbyte()`. For example, the function below sends a "note on" command on channel 1 with a key number and a velocity. To send a "note off", call this function with velocity zero (this is a special case in the MIDI protocol).

```
note_on(key, velocity)
    {
    sendbyte 0x90
    sendbyte key
    sendbyte velocity
    }

note_off(key)
    note_on key, 0
```

Other notes (loosely related to the MIDI-RS232 converter)

MIDI uses 16 channels. An instrument may be selected for any of these channels, but the MIDI standard does not say what instruments exist and what these sound like. Channel 1 may be programmed to sound like a piano or like a clarinet. This, of course, makes it impossible to create a MIDI file that sounds okay wherever you would play it. To alleviate this problem, an auxiliary standard, "General MIDI", describes 127 instruments with unique "patch numbers". This enables a MIDI file to select the appropriate patch numbers before sending the notes.

Sound cards often also provide a MIDI connector. On a sound card, however, the MIDI connector is often combined with a "game adapter" (for joysticks), using a single 15-pin sub-D connector. Pins 12 and 15 are MIDI transmit and MIDI receive respectively. These pins carry TTL logic signals, however, not the MIDI current loop. The cable that connects a sound card to a MIDI instrument contains electronics that converts between TTL logic and current loop.

Further reading & references

Programmable MP3-player for kiosk applications

Our product: a programmable MP3 player/controller, which provides a flexible RS232 port.

MIDI Protocol Guide

A description of the MIDI protocol, that focuses on the standard MIDI commands.

Professional MIDI Guide

A page from the same site as the above one, but this one focusing on the hardware design.

Fastcom 232/4-PCI-335, an RS232 adapter for non-standard baud rates

This four port RS232 adapter (PCI bus) supports non-standard baud rates up to 1.0 Mbps.

UC232R-10: Economy USB-RS232 Serial Converter Cable

The UC232R-10 is a low-cost USB-to-RS232 adapter that supports non-standard Baud rates.

https://www.compuphase.com/electronics/midi_rs232.htm