Ruin & Wesen MiniCommand Preliminary Manual

Table of Contents

Introduction

Thank you for choosing the Ruin & Wesen MiniCommand. The MiniCommand is a flexible MIDI controller designed to be small and easy to use. In fact, it is a very special MIDI controller, because you can completely change the way it functions: you can use it as a sequencer, as a controller, as a backup tool, as a MIDI merger, as a MIDI clock, etc… The principle behind this is very simple: a MIDI controller contains a small CPU (like the processor in your desktop computer) that runs a single program. This program controls everything the device does: it recognises button presses and knob turns, it displays data on the screen, it receives and sends MIDI messages, and it can read and store files on the internal SD-card. Normally, a MIDI controller has a single firmware that can't be easily changed (either the manufacturer has to upgrade it, or you have to use a complicated flashing tool), and you are limited to what the manufacturer has programmed into this firmware.

The approach of the MiniCommand is the exact opposite of this. You can very easily upload a new firmware over the MIDI connection to the device. Thus, you can easily browse the Ruin & Wesen Firmware repository, choose a firmware you like and want to try, upload it to the device, and get down to business. The MiniCommand needs to be connected to a computer just the time it takes to upload the new firmware. After that, you can use it as a standalone MIDI controller.

The firmware possibilities are endless. At the time this document is being written, most firmwares provided by Ruin & Wesen are oriented towards Elektron Machinedrum users, although we provide a simple MIDI Controller firmware as well. We provide a controller that allows for controlling the effect sections of the Machinedrum, that has a very easy MIDI learn functionality displaying parameter names, we have a polyrhythmic sequencer, we have a firmware providing backup snapshots every 30 seconds of the current pattern on the Machinedrum, etc…

We also provide a complete development environment for the MiniCommand, allowing users who want to program their own firmwares to very easily do so. The development environment is based on the Arduino environment, but comes with a full set of Ruin & Wesen libraries designed specifically for the MiniCommand. We provide a full featured MIDI stack, allowing you to use the following features:

- Sending and receiving all kind of MIDI messages (notes, CCs, aftertouch, pitchwheel, NRPN, sysex, etc…)
- Generating and synchronising to MIDI clock streams
- Sequencing events
- Sending and receiving samples over SDS

Firmwares written by users can very easily be uploaded to the online Ruin & Wesen firmware repository, where they can be accessed and modified by more users. We hope to launch a thriving firmware community for the MiniCommands, allowing programmers to share their creations and get feedback, and users who are not programmers to access a huge amount of custom firmwares and help with their development as well.

Of course, we know that many users don't have the time, the energy or the knowledge to develop their own firmwares, which is why we provide a very simple tool allowing you to browse the online patch repository, search it using simple search terms, and download firmwares provided by Ruin & Wesen and by other users. This way, you don't need to know anything about programming to try out the fantastic possibilities provided by this open source development model. You are then welcome to contribute your own ideas by sending feedback to the developers in the online forums at Ruin & Wesen.

In this document, we will describe the MiniCommand and provide a detailed description of the firmwares that come with it. Furthermore, the MidiDuino development environment which is used to program the MiniCommand is described, and a tutorial will guide you through the development of many small firmwares, putting light on different functionalities of the included MIDI and GUI libraries. These libraries can also be used with a normal Arduino board, allowing you to use the full functionality of the Ruin & Wesen MIDI stack with your own electronics.

Hardware Description

The MiniCommand comes in a solid die-cast hand-milled aluminium casing, making for a heavy, small, portable and comfortable device. It features a character LCD display, 4 control encoders (with push button functionality), 4 control buttons, a power-on slider switch and 2 status LEDs. It has a power connector (DC 7-10V, center plus polarity) and 3 MIDI connectors: INPUT 1, OUTPUT and INPUT 2. The microcontroller (the small CPU that runs the controller firmware) used in the MiniCommand is an Atmel AVR atmega64, with an additional 128 kB of RAM, and a microSD slot (containing a factory 1 GB microSD-card). There purposefully is no inscription on the MiniCommand because the functionality of these control elements depends on the currently loaded firmware (have a look at the chapter RW Firmwares describing the various firmwares by Ruin & Wesen).

The microcontroller comes loaded with a bootloader (a small program that is always executed first when the device starts) that allows the user to upload a new firmware over MIDI. To reprogram the MiniCommand, you can use either the MidiDuino development environment, which is presented in detail in this document, or the PatchDownloader which will soon be made available on the Ruin & Wesen website.

Changing the microSD-card

When you open the MiniCommand by removing the four screws on the bottom plate of the casing, you can access the internal microSD-card. While this card stores files in the common FAT32 format and can be read and written from your desktop computer, it is more meant as a kind of internal harddrive which only needs to be changed infrequently. If you need to swap the microSD-card, slide the metal holder carefully to the right, swap out the microSD-card, insert the new one carefully into the metal holder, and slide it back to the left. The microSD-card is pretty small and fragile, please be careful with it.

Connecting the MiniCommand to a MIDI device

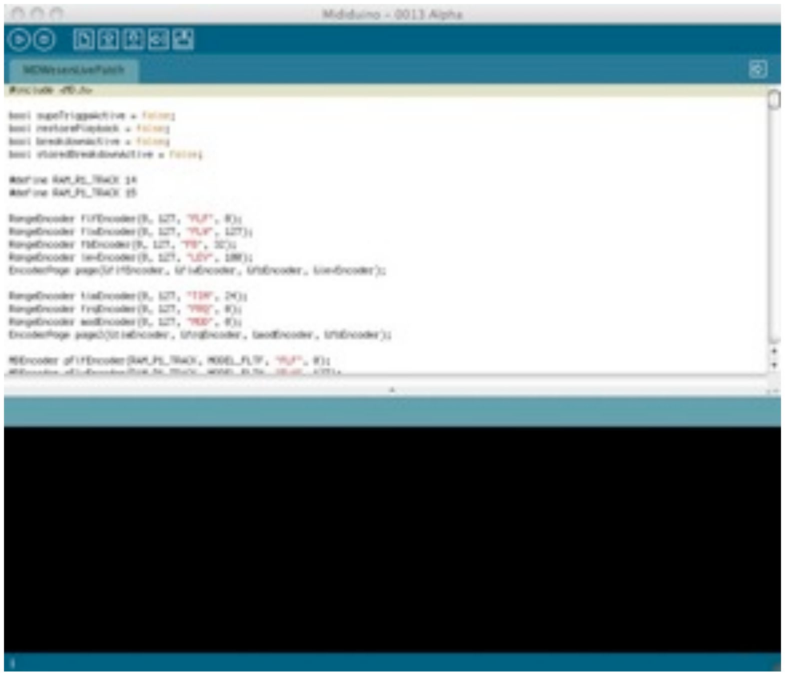
To connect the MiniCommand to a device, simply connect the MIDI OUTPUT of the device to the first MIDI INPUT of the MiniCommand (on the left), and the MIDI INPUT of the device to the MIDI OUTPUT of the MiniCommand (the MIDI connector in the middle). This way, the MiniCommand can communicate in both directions with the device, sending it data and receiving answers. The second MIDI INPUT on the MiniCommand may seem disturbing at first, but its purpose is to allow the MiniCommand to work as a merger, or to have an additional input (for example from a keyboard or from a second controller). One way to, for example, connect the MiniCommand to an Elektron Machinedrum, is to connect the output of the Machinedrum to the input of the MiniCommand (on the left), and the output of the MiniCommand to the Machinedrum.

In more elaborate setups multiple controllers and synthesizers are used, setup can become a bit more difficult (especially if a global clock has to be sent to multiple devices). One way to for example connect a MiniCommand to control a MachineDrum, which sends a MIDI clock signal to a MonoMachine is to use the Machinedrum as a master, connect its output to the input of the MonoMachine, use the THRU output of the monomachine and connect it to the input of the MiniCommand, and connect the output of the MiniCommand to the Machinedrum. This way, the MonoMachine receives a clock from the MachineDrum, the MiniCommand receives all the output of the MachineDrum (which is important because most patches react on incoming Sysex Data from the Machinedrum), and the MachineDrum is controlled by the MiniCommand. For more complicated setups, the MiniCommand can be used as a merger to merge two incoming MIDI streams. Furthermore, it can synchronize to a clock signal on either port, and retransmit it in sync on its output. This is also a very useful feature to “filter” an unstable clock signal (for example out of a computer) and retransmit it in a tight way. This functionality has however to be added by hand in the used firmware, there is no way to set it as a global setting for now (see below the section MIDI Clock programming).

The MiniCommand has no MIDI THRU, because the functionality of a MIDI THRU connector (which exactly replicates what is received of the input of a device) is mostly useful on a synthesizer, not on a controller. The purpose of a controller is to send messages on its MIDI OUTPUT, thus mirroring exactly its input serves no purpose. If you need to replicate a MIDI stream, the easiest is to use a simple MIDI buffer, which can either be purchased online (they are pretty cheap), or built with a simple transistor and optocoupler (see schematics on the Ruin & Wesen website).

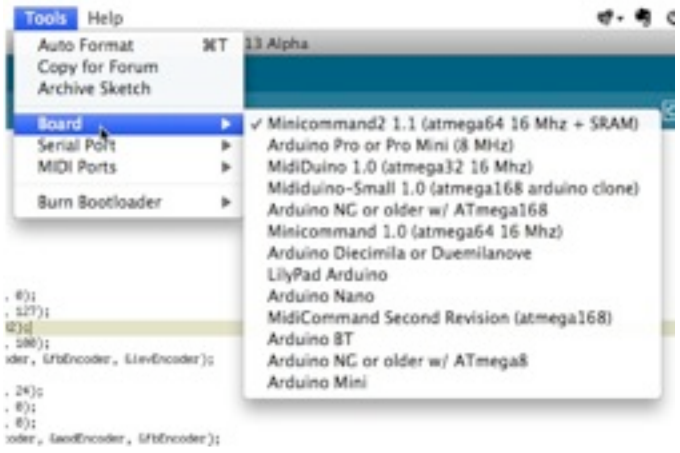
Software Installation

This chapter describes the installation of the MiniCommand development environment, which is called MidiDuino. You can download the MidiDuino environment for Windows and MacOSX from http://ruinwesen.com/mididuino . Unpack the zip file and copy the “mididuino-013” folder to your Applications folder, and start the “Mididuino 13.app” program. You will see the following screen.



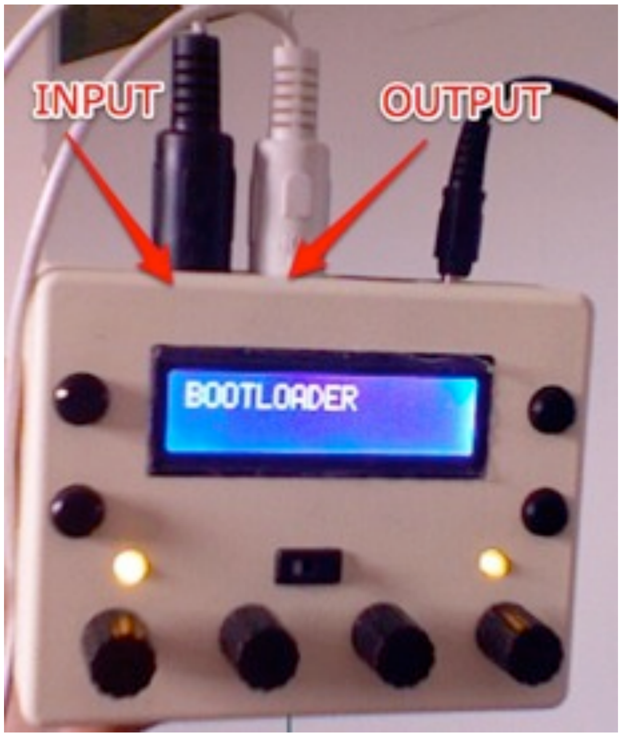
```
#include <MD.h>

bool supaTriggaActive = false;
bool restorePlayback = false;
bool breakdownActive = false;
bool storedBreakdownActive = false;

#define RAM_P1_TRACK 14
#define RAM_P2_TRACK 15

RangeEncoder flfEncoder(0, 127, "FLF", 0);
RangeEncoder flwEncoder(0, 127, "FLW", 127);
RangeEncoder fbEncoder(0, 127, "FB", 32);
RangeEncoder levEncoder(0, 127, "LEV", 100);
EncoderPage page(&flfEncoder, &flwEncoder, &fbEncoder, &levEncoder);

RangeEncoder timEncoder(0, 127, "TIM", 24);
RangeEncoder frqEncoder(0, 127, "FRQ", 0);
RangeEncoder modEncoder(0, 127, "MOD", 0);
EncoderPage page2(&timEncoder, &frqEncoder, &modEncoder, &fbEncoder);

MDEncoder pflfEncoder(RAM_P1_TRACK, MODEL_FLTF, "FLF", 0);
```

We now need to configure the device type that has to be programmed. Open the Tools -> Board dropdown menu, and select the “MiniCommand2 1.1” entry. This has to be done because Mididuino can be used to program all the future devices and development versions of Ruin & Wesen devices, or your own MIDI devices.

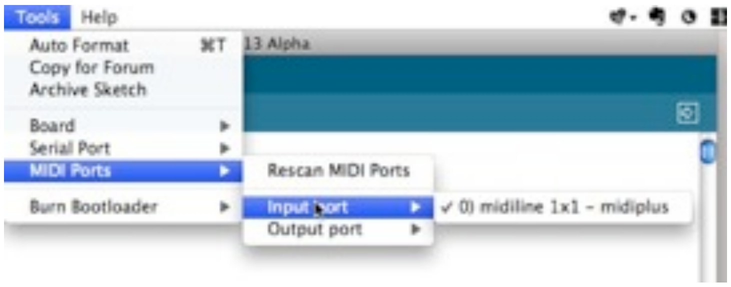

Tools
Help
13 Alpha
Auto Format ⌘T
Copy for Forum
Archive Sketch
Board
Serial Port
MIDI Ports
Burn Bootloader
✓ Minicommand2 1.1 (atmega64 16 Mhz + SRAM)
Arduino Pro or Pro Mini (8 MHz)
MidiDuino 1.0 (atmega32 16 Mhz)
Mididuino-Small 1.0 (atmega168 arduino clone)
Arduino NG or older w/ ATmega168
Minicommand 1.0 (atmega64 16 Mhz)
Arduino Diecimila or Duemilanove
LilyPad Arduino
Arduino Nano
MidiCommand Second Revision (atmega168)
Arduino BT
Arduino NG or older w/ ATmega8
Arduino Mini

Connect the MiniCommand to its power supply, switch it on, and connect it to a MIDI port on your computer. Connect the first INPUT of the MiniCommand (the port on the left when looking at the top of the MiniCommand) to your MIDI

OUTPUT on the computer, and connect the OUTPUT of the MiniCommand (the port in the middle, to the right of the first

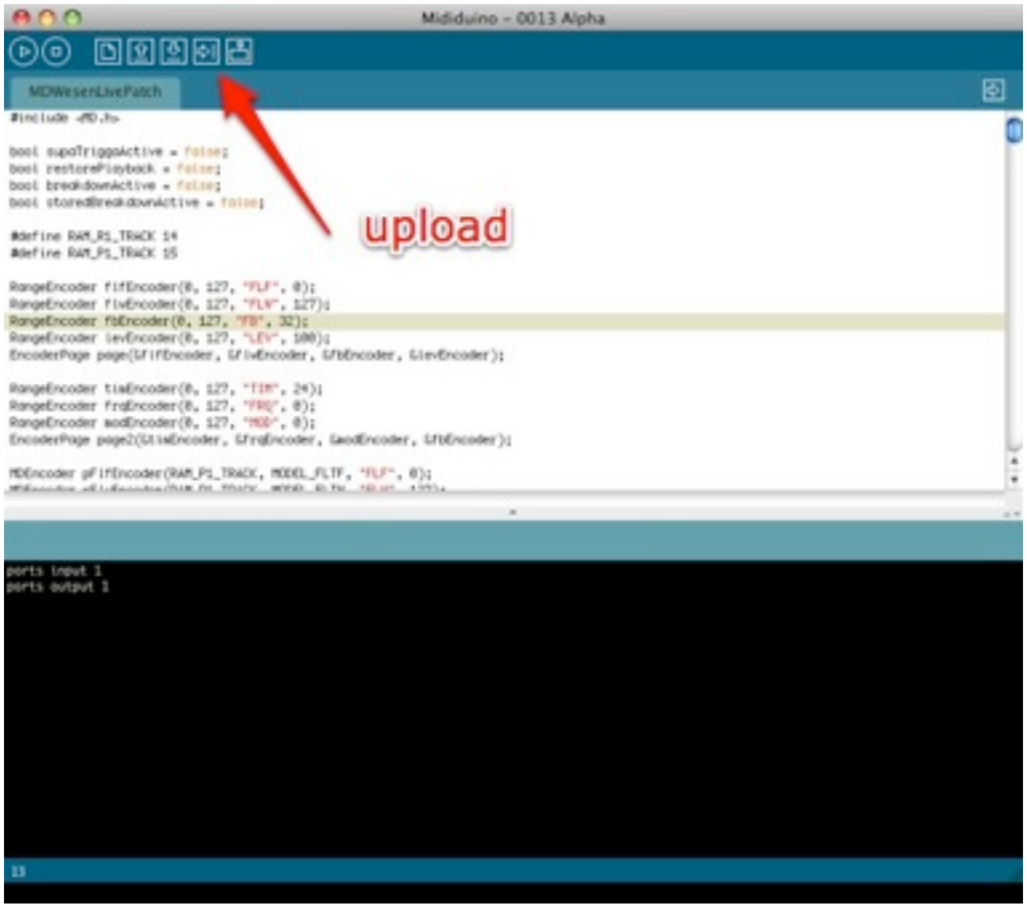


INPUT) to the MIDI INPUT on the computer.



Then open the Tools -> MIDI Ports dropdown menu, and select the appropriate MIDI INPUT port and MIDI OUTPUT port (you may need to use “Rescan MIDI Ports” if you connected your MIDI connector after opening the Mididuino program.

We now need to choose the firmware we want to upload. You can download a starting selection of Ruin & Wesen firmwares (which are also called sketches) from http://ruinwesen.com/static/media/MiniCommand-sketches.zip . You can either copy those sketches into your local sketch directory, which can be configured from the Preferences menu of the Mididuino program, or open it directly using the File -> Open menu. Once you have opened a firmware, the sourcecode of the firmware will be shown in the editor window of Mididuino. Don’t be afraid by it, we are not going to change anything. Assuming you want to upload the MDWesenLivePatch firmware, which is the firmware used by Wesen when playing live, the window should look like this:



```
#include <MD.h>

bool supoTriggoActive = false;
bool restorePlayback = false;
bool breakdownActive = false;
bool storedBreakdownActive = false;

#define RAM_R1_TRACK 14
#define RAM_P1_TRACK 15

RangeEncoder flfEncoder(0, 127, "FLF", 0);
RangeEncoder flwEncoder(0, 127, "FLW", 127);
RangeEncoder fbEncoder(0, 127, "FB", 32);
RangeEncoder levEncoder(0, 127, "LEV", 100);
EncoderPage page(&flfEncoder, &flwEncoder, &fbEncoder, &levEncoder);

RangeEncoder timEncoder(0, 127, "TIM", 24);
RangeEncoder frqEncoder(0, 127, "FRQ", 0);
RangeEncoder modEncoder(0, 127, "MOD", 0);
EncoderPage page2(&timEncoder, &frqEncoder, &modEncoder, &fbEncoder);

MDEncoder pFlfEncoder(RAM_P1_TRACK, MODEL_FLTF, "FLF", 0);
```

```
ports input 1
ports output 1
```

Now all we need to do is press the upload button which is shown in the above screenshot. This will upload the firmware to the device over sysex. The MiniCommand should reset itself and enter the BOOTLOADER menu, which displays "BOOTLOADER" on the screen (as shown in the picture above). If the MiniCommand doesn't enter the bootloader automatically on receiving MIDI, make sure that the MIDI connections are correct. You can also enter the bootloader manually by turning off the unit, holding down the button on the upper right, and switching on the unit while still holding down the button. The MiniCommand sends a 6 bytes MIDI Sysex message when entering the booloader (you can use the MIDI Monitor tool from http://www.snoize.com/MIDIMonitor/ to see if the MiniCommand answers Sysex messages. If the monitor doesn't show a 6 bytes answer, then probably the MIDI connections are wrong). When receiving a firmware, the bootloader shows the number of received blocks, and starts the new firmware when it is completely received.

XXX screenshot midi monitor

If something goes wrong during uploading, the bootloader will display "WRONG CHECKSUM", and the best thing to do is to check if your MIDI cables are broken and try to upload again (some USB-MIDI interfaces sometimes have trouble with sysex as well). The bootloader can never be destroyed by uploading incorrect firmwares, so that you always have a safe way to reupload a new firmware if something goes wrong.

RW Firmwares

This chapter gives a short introduction to the different firmwares provided with the MiniCommand. They are mostly geared towards Elektron MachineDrum users. In the future, much more firmwares will be available on the Ruin & Wesen website. Also, these are mostly firmwares that we use while playing live and in the studio, so that they are tailored towards our needs and concepts. If you are a programmer, you are welcome to modify the sourcecode of these firmwares and customize them according to your preferences. If you are not into programming, you are welcome to share your ideas with us by sending us a message at info@ruinwesen.com , and by using the forums that will soon be online.

The actual version of our firmwares is avaible at our website on http://ruinwesen.com/mididuino , and you can also check out the code repository for mididuino at http://mididuino.googlecode.com/ .

Some of these sketches trigger the MachineDrum (namely MDNotes, MDArpeggiator and MDWesenLivePatch). To be able to use this functionality, the standard note to track mapping has to be set up on the MachineDrum in GLOBAL -> MIDI -> MAP EDITOR: C2 triggers 01-BD, D2 trigger 02-SD, etc... This will be changed in a future version of the library that will parse the complete GLOBAL data.

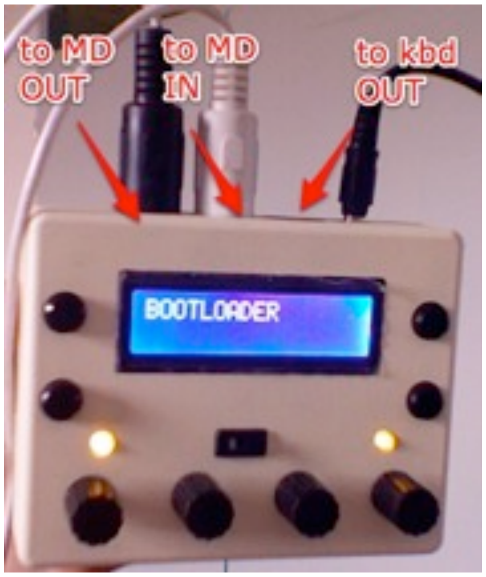
MDMonster

The MDMonster firmware is the firmware loaded by default on the MiniCommand. It combines all the firmwares presented in this chapter. When starting the MDMonster firmware, you can select the firmware to be run by turning the leftmost encoder. The name of the selected firmware will appear on the screen. To start the selected firmware, press the upper left button. You can return to the selection screen at any time by holding down all four buttons simultaneously, and select a new firmware by turning the leftmost encoder. Most of the parameter setting for each subfirmware will stay the same, even if firmwares are switched in between. So that if you have a nice groove going on in the sequencer firmware for example, switching to the randomizer and then coming back to the sequencer will not affect its settings.

MDNotes

The MDNotes firmware transforms your MachineDrum into a powerful and versatile monophonic/polyphonic synthesizer. Connect the MachineDrum to the INPUT and OUTPUT of the MiniCommand, and connect a MIDI Keyboard or another sequencer to the second MIDI INPUT of the MiniCommand. Notes received from the keyboard are converted into special messages for the Machinedrum that will allow you to play melodies. Some of the machines (for example EFM-SD, EFM-HH, TRX-BD, TRX-B2, ROM machines, etc…) can actually produce pitched sounds. The mapping from normal notes (for example C3, Eb5, etc…) to the pitches of these machines however is not very straightforward. What MDNotes does is take the incoming pitch of notes, and convert it into the correct pitch settings on the Machinedrum. Each channel of incoming MIDI notes is mapped to a specific sound on the Machinedrum. This can be:

- A monophonic mapping, that is that each note received from the keyboard will trigger one track on the Machinedrum,
- A polyphonic mapping, that is notes received from the keyboard are distributed to multiple tracks on the MachineDrum with the same machine loaded, allowing you to play chords and complex melodies,
- A layered monophonic (or multiphonic) mapping, where each note received from the keyboard will trigger multiple tracks with the same pitch, allowing you to layer a TRX-BD under a sample for example.



The first thing that has to be done after connecting MDNotes to the MachineDrum is to load the currently selected kit. Press the button on the upper left to reload the kit of the MachineDrum (both IN and OUT have to be connected to the MachineDrum). Once the Sysex data describing all the parameters of the kit has been loaded, the MiniCommand displays the name of the currently loaded kit.

You can now configure the MDNotes settings by pressing the button on the lower left, which will allow you to chose a MIDI channel to configure. Press the pad corresponding to the desired channel on the Machinedrum (BD -> channel 1, SD -> channel 2, etc…). This selects which of the incoming MIDI channel is going to be configured. You will be shown with the following display, which shows which track of the machinedrum is loaded with a melodic machine (a machine that can produce pitched sounds). A _ signals that a track doesn't have a melodic machine loaded and can thus not be used by MDNotes.

The rightmost encoder selects the type of mapping used for the selected incoming MIDI Channel.

"NML" is the normal monophonic mapping, in which the incoming MIDI note on channel X is sent to the Machinedrum Track X (the machine type of the selected track is shown on the MiniCommand). If there is no melodic machine loaded on that track, the MiniCommand shows a XXX.

"PLY" is the polyphonic mode. Using the step-buttons on the Machinedrum, you can select which track on the machinedrum is going to be active in the polyphonic mode. A selected track is going to be used in a cycle to play chord notes, and is shown by a "X" on the MiniCommand screen. A non-selected track is shown by a ".". Each note received by the MDNotes will go to the next track in the selected tracks, so that it "cycles" through active machines. If a parameter on one machine is modified on the Machinedrum (for example FLTF), the MiniCommand will forward that parameter modification to all the active tracks, so that the overall sound of the polyphonic synthesizer is the same on each note.

"MLT" is the multiphonic mode. Similar to the polyphonic mode, you can select which sounds will be layered when a note is hit on the keyboard. However, each machine is triggered by each received notes, so that you can't play chords with the multiphonic machine. However, you can use it to layer sounds in interesting ways. Furthermore, parameter changes are not broadcast like in the polyphonic mode.

If you modify the current kit by changing machines, you need to save the kit and reload it on the MiniCommand by pressing the upper left button, else the MiniCommand doesn't notice the machine change and may display wrong information and send the wrong MIDI parameters to the MachineDrum.

MDMelodyHelper

This firmware is a simple tool to help you input melodic data on the MachineDrum. Some of the machines on a MachineDrum (for example EFM-HH or ROM machines) can play melodic sounds, and thus be used to input chords and notes. However, the mapping from PTCH parameter to actual sounding pitch is not always very easy, and inputting longer and more complicated melodies gets quite tedious quickly.

MDMelodyHelper has to be connected to the MachineDrum using INPUT and OUPUT. Like for the MDNotes firmware, you first need to load the current Machinedrum kit on the MiniCommand by pressing the upper left button. MDMelodyHelper will then display the name of the currently loaded kit. When you know turn the PTCH parameter on a melodic machine, MDMelodyHelper will display the actual sounding pitch of the machine. As often pitches fall in between real pitches, it will display a number of "+" signs the further away the sounding pitch is from the actual pitch. This way, you can quickly tune all the notes in a melody to the actual pitch without having to look up parameter values in long boring tables.

MDWesenLivePatch

This firmware is a mixture of different firmwares, and it is the one used by Wesen when he plays live with the MachineDrum. You need to connect the MiniCommand to the MachineDrum using the first MIDI INPUT and the MIDI OUTPUT. The MachineDrum needs to send out MIDI tempo information so that the MiniCommand can synchronize itself. MDWesenLivePatch has two main functions.

The first function of MDWesenLivePatch is controlling the delay effect of the Machinedrum, and the RAM-P1 playback track (loaded on track 16. Wesen uses the RAM-R1 to resample the currently playing pattern, and plays it back on track 16 when switching over to the next pattern). These are simple MIDI controller pages. To switch the controller page, hold down the upper left button, and press any other button to switch the page. The page selected by:

- the bottom left button is the first delay control page, controlling FLTF, FTLW, FB and LEV of the master delay effect. This is very useful to do extensive dubbing and ambience noises with the delay, as you can put the delay into full feedback (FB > 64), and quickly turn it back down when it threatens to get too loud. This used in conjunction with the filter frequencies allows you to create noise buildups, pads, rumbling basses, and a host of different other sounds.
- the bottom right button is the second delay control page, controlling the TIM, FRQ, MOD and FB of the delay. This allows you to create weird rythmic noises and effects by modifying the time modulation of the delay, which will then pitch up and down as well, as it is modelled like a tape delay.
- the upper right button controls the RAM-P1 parameters, and is very useful to do DJ-like bass cuts or high cuts while mixing the resampled pattern with the new pattern. This allows you to mix in new sounds on the Machinedrum while controlling the resampled version from the MiniCommand.

The second function of MDWesenLivePatch is much more elaborate and interesting. MDWesenLivePatch is tempo-synchronized to the MachineDrum, and automatically sends triggers to the RAM-P1 machine, so that no trigger has to be kept on the MachineDrum (track 16, with the RAM-P1, should always be empty of hits). To use this, the MachineDrum has to be configured to send out tempo and control messages, and the note D4 on channel 1 has to trigger the RAM-P1 track. In a future version of the firmware these settings will be automatic, at the moment it's kind of tailored to Wesen's live setup. This allows the MiniCommand to cut up the playing loop. Hold down the upper left button, and select the length of the repeated pattern with the first encoder on the left. The default is two bars (the standard length on the MK1 Machinedrum), but you can pull that down to 16th, if you want some gabba. Furthermore, this screen allows you to put in “breakdowns” which is slight variations at the end of certain bars. You can select the frequency of breakdowns with the third encoder from the left.

When you release the upper right button, the breakdown mode is no longer active and the pattern plays back normally. However, you can lock the breakdown mode by holding down the upper right button, and pressing the upper left button. The MiniCommand will show “BREAKDOWN ON”, which means that breakdown and repeats will still be active when the upper right button is released. To turn off the breakdown mode, hold down the upper right button and press the upper left button. The MiniCommand will show “BREAKDOWN OFF”.

Finally, the last and coolest feature of MDWesenLivePatch is the inbuilt SupaTrigga reverser, which will slice up the playing loop in cool and interesting ways. Just hold down the lower right button, and let the loop play. As long as the button is held down, the resampled pattern will be sliced in rythmically meaningful ways, with reverses, repeats, etc… When the button is released, the resampled pattern is played back normally.

MDRandomize

The MDRandomize firmware is an “intelligent” randomizer for the MachineDrum. Like with most other firmwares, you connect the MiniCommand to the MachineDrum using MIDI INPUT and MIDI OUTPUT, and you need to load the current kit of the MachineDrum by pressing the upper left button. The name of the kit will be displayed in the upper left of the MiniCommand display. This will initialise the starting value of all the parameters.

You can then select the track to be randomized by turning the leftmost encoder. The machine type will be displayed in the upper line of the MiniCommand. The “intelligence” of the MDRandomize firmware comes from the fact that you can select the range of the parameters to be randomized by turning the first encoder. It has the following modes:

- “FILTER” will randomize FTLF, FTLF and FTLQ,
- “AMD” will randomize AMD and AMF,
- “EQ” will randomize EQF and EQG,
- “UPSYN” will randomize the first four parameters (the upper row of synthesizer parameters),
- “LOWSYN” will randomize the lower four parameters (the lower row of synthesizer parameters),
- “SYN” will randomize the first eight parameters (the first page),
- “LFO” will randomize the LFO settings,
- “SENDS” will randomize the DEL and REV parameters,
- “DIST” will randomize “SRR” and “DIST”,
- “FX_LOWSYN” will randomize like “FILTER”, “AMD”, “EQ” and “LOWSYN” combined,
- “FX_SYN” will randomize like “FILTER”, “AMD”, “EQ” and “SYN” combined,
- “ALL” will randomize all parameters.

This way, you can select the range of parameters to be randomized. The randomization amount can be set by the third encoder from the left. When you press the bottom left button, the parameters will be randomized. To undo a randomization, press the lower right button. Also, if you turn a parameter value on the MachineDrum, future randomization will use that value as a starting point, allowing you to tweak parameters as you go exploring.

Further versions of this firmware will allow custom parameter combinations depending on the loaded machine.

MDArpeggiator

The MDArpeggiator firmware is based on the same functionality as the MDNotes firmware. You need to connect the MachineDrum to the first MIDI INPUT and the MIDI OUTPUT of the MiniCommand, and you connect a keyboard to the second MIDI INPUT. The MachineDrum has to send MIDI Clock information to the MiniCommand so that both devices are synchronized, and has to receive triggers notes on the first channel (using the default MachineDrum trigger configuration). In a future version of the firmware, these settings will be automatic. You can then reload the currently loaded kit of the Machinedrum by pressing the upper left button. The MiniCommand will then display the name of the current kit.

The MiniCommand now works as a MIDI arpeggiator. When the MachineDrum's playback is started, the MiniCommand is synchronized to the tempo, and notes received from the keyboard are arpeggiated to the MachineDrum. You can select the destination track by turning the left-most encoder (which is called "TRK"). When you turn this encoder, the MiniCommand will display the name of the loaded machine on that track (or display "XXX" if the machine is not a melodic machine). You can set the speed of arpeggiation by turning the second encoder. A speed of 1 means that every 16th note is played by the arpeggiation. A value of 2, that every 8th note is played, etc… The number of octaves spanned by the arpeggiation can be set with the third encoder (named "OCT"). Sometimes the notes played by the arpeggiation are above or below the range of the target melodic machine. In those cases, the MiniCommand doesn't send any note. Finally, the rightmost encoder (named "LEN") sets the number of repetitions of the arpeggiation when a chord is hit on the keyboard. This can be an infinite number of arpeggiation, of up to 16. Of course, the arpeggiator stops playing when the notes are released on the keyboard.

The MDArpeggiator firmware is modelled after the Ableton Live Arpeggiator module. Further parameters of arpeggiation can be set on the second page. To switch to the second page, just press the bottom left button. To go back to the first page, press the upper left button. On the second page, you can set the arpeggiation style, which is the same as the arpeggiation style in Ableton: "UP", "DOWN", "UPDOWN", etc… You can also set the arpeggiation trigger, which resets the sequence. When it is "OFF", the sequence is retriggered every time a new note is hit on the keyboard when all previous notes were released. You can also retrigger the arpeggiation on every new note by setting "TRG" to "NOTE". Finally, you can retrigger at repeated rhythmic intervals by setting "TRG" to "BEAT" and selecting the number of 16th notes between retriggers by using the rightmost encoder.

MDpitchEuclid

This firmware is a very interesting experiment. It generates algorithmic basslines on the MachineDrum. You need to connect the MachineDrum to the first MIDI INPUT and the MIDI OUTPUT of the MiniCommand, and you connect a keyboard to the second MIDI INPUT. The MachineDrum has to send MIDI Clock information to the MiniCommand so that both devices are synchronized. Once the MiniCommand is turned on, reload the current MachineDrum kit by pressing the upper left button. The MiniCommand will display the name of the currently loaded kit.

This firmware will generate a polyrhythmic bassline on the MachineDrum by sending pitched notes to it (like MDNotes or MDArpeggiator). To select the destination melodic track, hold down the bottom left button, and turn the first encoder. The name of the loaded machines will be shown on the MiniCommand display, or "XXX" if the selected track doesn't have a melodic machine. Once a track is selected, you can start the playback of the generated bassline by pressing play on the Machinedrum (and make sure it sends tempo information to the MiniCommand).

The bassline is generated by using an underlying euclidean rythmic pattern. This pattern is generated by distributing a number of pulses (set by the "PLS" encoder) over a length of time (set by the "LEN" encoder and counted in 16th notes). These pulses are distributed in the most "equidistant" way possible. For example, distributing 3 pulses over 9 notes gives the pattern “X . . X . . X . .” while 3 pulses distributed over 8 notes gives the pattern “X . . X . . X .” . This generates very natural sounding rhythmical patterns. The “OFF” encoder sets the initial displacement of the rhythmical pattern.

Over these hits, MDpitchEuclid distributes a number of pitches chosen from a Cminor7 arpeggio. The number of notes in the bassline is chosen by the “PTC” encoder, and the pitches themselves are randomized by pressing on the first encoder. Thus, you can have a different number of notes for the bassline than the number of hits in the repeating rhythmical pattern. You can create quite complex circling basslines very quickly by just adjusting a few knobs on the MiniCommand.

MidiDuino and the MiniCommand

The difference between what we call MidiDuino and the MiniCommand can be a bit confusing. MidiDuino is on the one hand a modification of the Arduino environment geared towards MIDI devices, and also a set of libraries that make working with MIDI on a microcontroller much easier. These libraries take care of handling the sending and receiving of MIDI messages, handles internal and external synchronisation and sequencing, as well as implementing a set of helper functions to modify MIDI data. They can also be used (although in a slightly restricted version) with normal Arduino boards as well, and can be found at http://ruinwesen.com/mididuino .

The environment used to program the MiniCommand is based on the Arduino platform. The Arduino platform is a physical computing design environment that allows designers, artists and tinkerers to build electronics project in a very simple manner. It's based on a small board (which exists in lots of different versions) that features a small processor (the microcontroller), and a number of pins to interface to the external world. It also has a software part which is used to write programs that are then uploaded to the board and run. This is very similar to the MiniCommand, except that the MiniCommand is a finished device that provides MIDI inputs and output, and an interface aimed at electronic musicians (the 4 encoders, buttons, and display). However, the goal of the MiniCommand is also to provide a simple platform for musicians, artists, designers and tinkerers to play and interact with MIDI data and MIDI devices (synthesizers, grooveboxes and sequencers). Being able to program a new firmware for your MIDI controller, and being able to program the device itself in a matter of seconds is immensely liberating, even if you already were used to program microcontorllers and build your own MIDI controllers the "old-fashioned" way.

The MiniCommand uses the same editing software to program the MiniCommand, which happens to have an AVR Atmega64 microcontroller that is of the same family as the AVR controllers used in the different Arduino boards. This editing software was modified to provide MIDI support. The normal Arduino board uses a serial USB interface to communicate with the computer, while the MiniCommand is programmed directly over MIDI. Furthermore, the editing software has been slightly modified so that the integration of the Ruin & Wesen MIDI libraries is easier. It may be worthwhile to note that the software running on the MiniCommand does not use the normal Arduino libraries at all, but provides specially optimized code for the device.

So, to recapitulate, the MiniCommand is the device you are holding in your hands, and MidiDuino is both the development environment used to program firmwares for the device, and a set of libraries providing support for MIDI communication.

Programming the MiniCommand: a short tutorial

This chapter will give you a smooth introduction into programming firmwares for the MiniCommand using the MidiDuino environment. You first need to set up the MidiDuino environment as described in the first chapter of this manual. Once you are able to upload new firmwares, you are also ready to develop your own. Open the MidiDuino editor, and create a new empty patch. The programming language used to write firmwares for the MiniCommand is basically the Arduino language, which basically is C++. If you have never written a program in Arduino or C++, we encourage you to take a look at the Arduino reference pages under http://www.arduino.cc/en/Reference/HomePage , and sheepdog's programming tutorials at http://sheepdogsoftware.co.uk/pltut.htm . We are planning to provide an extensive programming tutorial aimed especially at musicians.

Hello World

The empty file you have open in MidiDuino is our blank slate on which we are going to create our firmware. This file (which we also call a "sketch") is going to be compiled by the MidiDuino environment, and then uploaded to the MiniCommand. Every little thing the firmware is going to handle is going to be written in this file. However, this isn't scary at all, in fact, most of the complicated stuff like receiving messages, writing things on the screen, recognizing encoder turns and button presses, is already handed by the supporting MidiDuino libraries.

Without further talk, here is our first sketch, which displays "HELLO WORLD" on the display of the MiniCommand.

```
void setup() {
      GUI.put_string_fill("HELLO WORLD");
}

void loop() {
      GUI.updatePage();
      GUI.update();
}

void handleGui() {
}
```

The only thing that is required in a sketch is that you provide 3 functions that constitute the backbone of the sketch. These three functions are: **setup()**, **loop()** and **handleGui()**.

The **setup()** function is called at the startup of a device (or after a reset), and is executed once before everything else in the sketch. This is the place to initialize variables and functions, and for example to setup MIDI sequencing and synchronization. In this first sketch, we call a function of the GUI library to display the string "HELLO WORLD". All the functions of the GUI library are used to either display text and numbers on the display (in an organized way, as we will see in the next few sketches), or to read the GUI status (how much an encoder has been turned, if a button has been pressed, etc). Furthermore, the GUI library implements the concept of pages: a page is a grouping of up to 4 encoders that are displayed on the LCD. The GUI library allows for easy switching of pages, easy naming of parameters, and also the reuse of the same parameters on different pages.

The **loop()** function is called in a loop (as its name says) at regular intervals for the whole runtime of the software. This is where you can check for GUI changes (for example if an encoder has been turned, etc...) and react accordingly. The **loop()** function in this sketch calls two further functions from the GUI library which are used to refresh the current graphic state of the LCD. **GUI.updatePage()** is used to refresh the state of the current page and to refresh the current encoder counters, and needs to be called right at the beginning of the loop() function before any encoder handling code (which we will see in the next few sketches) is used. **GUI.update()** is used to display the update information on the display, and needs to be called at the end of the **loop()** function.

Finally, the **handleGui()** function is a bit special. It's similar to **loop()** in that it is called regularly through out the runtime of the firmware. However, it is called out of an interrupt routine that is called a few thousand times per second, and polls the status of the buttons. This means that a timer interrupts everything that is happening on the MiniCommand, reads in if buttons have been modified, and calls the **handleGui()** function before resuming the normal execution of the software. Because of this, code in the **handleGui()** has to be very short in order not to interfere with the normal execution of the firmware, and it needs to take care when accessing variables, as these can be accessed from the "normal" executing code as well. The reason for this complicated setup is that we need to recognize button presses (and not just if a button is down or up) immediately after the press happens. If you find that your sketch is responding in a weird fashion, make sure that the code in **handleGui()** only does what is necessary. This whole construction will be replaced by a slightly more elegant event queue handling in a future version of MidiDuino.

Hello World with buttons

In this next sketch, we are going to recognize button presses and button status, to highlight the different uses of the GUI and the **handleGui()** function. The status of the buttons can be queried by using macros from the GUI library (you can check the complete API reference at the end of this document). The status of the buttons itself is stored in different variables that are called **Buttons.BUTTON1** (the button on the upper left), **Buttons.BUTTON2** (the button on the lower left), **Buttons.BUTTON3** (the button on the lower right) and **Buttons.BUTTON4** (the button on the upper right). The individual status of the encoder buttons can also be queried, although it is often handled automatically by incrementing turns by 5 rather by 1 (like on the MachineDrum). The macros used to query the button statuses are **BUTTON_DOWN()** to check if a button is held down, **BUTTON_UP()** to check if a button is released, **BUTTON_PRESSED()** to query if a button has been pressed, and **BUTTON_RELEASED()** to check if a button has been released. **BUTTON_PRESSED()** and **BUTTON_RELEASED()** can only be used in the **handleGui()** function, because they rely on being called immediately after the input polling.

In the first sketch with buttons, we are going to use only **BUTTON_DOWN()** and **BUTTON_UP()** in the **loop()** function. This way of doing the polling is not very efficient, because the display will be rewritten on every call to **loop()**, which is why there is a call to delay to avoid flooding the display and making it non responsive.

```
void setup() {
}

void loop() {
      GUI.updatePage();
      if (BUTTON_DOWN(Buttons.BUTTON1)) {
            GUI.put_string_fill("DOWN");
      } else {
            GUI.put_string_fill("UP");
      }
      GUI.update();
      delay(10);
}

void handleGui() {
}
```

A much better way to handle this is to use **BUTTON_RELEASED()** and **BUTTON_PRESSED()** in the handleGui() function. The call to **GUI.put_string_fill()** just writes into a memory buffer, and thus is very fast. The actual sending of the data to the display is handled by **GUI.update()**, which takes much more time. This code will write something new to the screen only on the press or the release, not in between.

```
void setup() {
}

void loop() {
      GUI.updatePage();
      GUI.update();
}

void handleGui() {
      if (BUTTON_PRESSED(Buttons.BUTTON1)) {
            GUI.put_string_fill("PRESSED");
      }
      if (BUTTON_RELEASED(Buttons.BUTTON1)) {
            GUI.put_string_fill("RELEASED");
      }
}
```

The first encoder page

In this sketch, we are going to going to create our first controller page. A page displays up to four encoders on the screen. Each encoder gets 4 characters, as the screen can display 2x16 characters. The first line displays the name of the encoders, the second line displays the value of the encoders. These encoders are "virtual" encoders, and you can have as many as you want in a sketch. They are then linked to the actual hardware encoders when a page becomes active. You can think of virtual encoders as parameters in your firmware, that are controlled by the hardware encoders when they are made “active”. This allows you to have the same encoder on different pages for example, or to update virtual encoders when receiving MIDI parameters, or when a sequenced event is taking place.

The GUI library handles the updating of the encoders and the displaying of changed values. The actual hardware encoders are handled separately and automatically changed when the hardware knobs are turned or pressed. However, we need to update the “virtual” encoders which are stored in the page. This is done by the **GUI.updatePage()** call at the beginning of every loop. This will copy the movements of the hardware encoders into the virtual encoders, and flag virtual encoders which have changed.

There are different types of virtual encoder classes, the toplevel one (with no internal functionality) being the **Encoder** class. The most useful for working with MIDI is the **RangeEncoder** class, which puts a lower boundary and an upper boundary of the possible values of the encoder. The most useful instance thus is **RangeEncoder(0, 127)**, which limits the range to the range of MIDI CC parameters. In addition to the limits, each encoder can be given a name that is displayed automatically by the GUI library. In the sketch, we define two parameters by using the following lines of code:

```
RangeEncoder param1(0, 127, “P1”);
RangeEncoder param2(0, 127, “P2”);
```

After the encoders have been define, we need to create the actual page. There is a toplevel class **Page** which can be overriden for custom page behaviour. We however use a simple subclass called **EncoderPage** which can take up to four encoder references at its creation, and displays those automatically. The page is created with the following line:

```
EncoderPage page(&param1, &param2);
```

The GUI library then needs to be told that this page is going to be the active page, using the **GUI.setPage()** function. This function is called in the setup, as is shown in the complete code of the sketch:

```
RangeEncoder param1(0, 127, “P1”);
RangeEncoder param2(0, 127, “P2”);
EncoderPage page(&param1, &param2);

void setup() {
      GUI.setPage(&page);
}

void loop() {
      GUI.updatePage();
      GUI.update();
}

void handleGui() {
}
```

Of course, this whole sketch can be extended to have two pages very easily. We also add two new encoders into the mix to make things more interesting. The pages are switched when **Buttons.BUTTON1** or **Buttons.BUTTON2** are pressed.

```
RangeEncoder param1(0, 127, “P1”);
RangeEncoder param2(0, 127, “P2”);
RangeEncoder param3(0, 127, “P3”);
RangeEncoder param4(0, 127, “P4”);

EncoderPage page(&param1, &param2);
EncoderPage page2(&param3, &param4, &param2);

void setup() {
      GUI.setPage(&page);
}

void loop() {
      GUI.updatePage();
      GUI.update();
}

void handleGui() {
      if (BUTTON_PRESSED(Buttons.BUTTON1)) {
            GUI.setPage(&page);
      } else if (BUTTON_PRESSED(Buttons.BUTTON2)) {
            GUI.setPage(&page2);
      }
}
```

As you can see when trying out this firmware, page changes can be very easily done. Also, **param2** is used on both pages, and changing its value on one page will change the value on the other page as well. You can try to experiment with various ways of changing pages. For example, a very useful way to switch pages is to have a temporary “overlay” page which is only there while a button is held down. To do this, you can use the following **handleGui()** function:

```
void handleGui() {
      if (BUTTON_PRESSED(Buttons.BUTTON1)) {
            GUI.setPage(&page2);
      } else if (BUTTON_RELEASED(Buttons.BUTTON1)) {
            GUI.setPage(&page);
      }
}
```

Sending MIDI

Now this is all very nice and funky, but the controller still doesn't do anything useful. In this sketch, we are going to send and receive CC messages to control a simple encoder page. When a knob is turned, the sketch will send MIDI CC messages, and when it receives CC messages, it will update the values of the corresponding encoder.

Each virtual encoder can be checked for changes by calling the **encoder.hasChanged()** method, which returns true if the encoder has been changed by the **GUI.updatePage()** method. It is very important that **encoder.hasChanged()** gets called between **GUI.updatePage()** and **GUI.update()**, as **GUI.update()** resets the status of all the encoders. The current value of a virtual encoder can be read by calling the method **encoder.getValue()**.

When the status of an encoder has changed, we send its data over MIDI. For this, we use the extensive MIDI library that comes with MidiDuino. The MIDI library is split in two parts. One part takes care of the physical MIDI interface, which is an asynchronous serial interface, often called UART in electronics. This is mostly used to send out MIDI data. The MiniCommand uses the object called **MidiUart** to represent the first MIDI port, which has both an input (leftmost MIDI connector), and an output (middle MIDI connector). It also has a second object called **MidiUart2**, which is only used for input (rightmost MIDI connector). The **MidiUart** has a long list of functions for every kind of MIDI message. We are going to send out CC messages using the **MidiUart.sendCC()** function. This function can take two or three arguments, depending on if the standard MIDI channel is to be used, or a specific channel is used. The default channel can be set by setting the variable **MidiUart.currentChannel**.

In the following sketch, the first encoder is sent as CC 1, and the second encoder is sent as CC 2.

```
RangeEncoder param1(0, 127, “P1”);
RangeEncoder param2(0, 127, “P2”);
EncoderPage page(&param1, &param2);

void setup() {
	GUI.setPage(&page);
}

void loop() {
	GUI.updatePage();
	if (param1.hasChanged()) {
		MidiUart.sendCC(1, param1.getValue());
	}
	if (param2.hasChanged()) {
		MidiUart.sendCC(2, param2.getValue());
	}
	GUI.update();
}

void handleGui() {
}
```

It's very simple to extend the controller code by for example adding code to send out notes when a button is pressed, which is done by modifying the **handleGui()** function like this.

```
void handleGui() {
	if (BUTTON_PRESSED(Buttons.BUTTON1)) {
		MidiUart.sendNoteOn(1, 100);
	}
	if (BUTTON_RELEASED(Buttons.BUTTON1)) {
		MidiUart.sendNoteOff(1);
	}
}
```

Receiving MIDI

Receiving MIDI on the MiniCommand is very easy. The incoming serial data on **MidiUart** and **MidiUart2** is passed automatically to the MIDI stack, which parses the incoming messages, taking care of things like running status, realtime messages interrupting the data, and handling Sysex data (actually handling Sysex data is a bit more complicated, and will be covered in a separate tutorial). Once an incoming message is recognized, the MIDI stack checks to see if a callback function has been registered. If this is the case, the callback is called with the received message. So in order to receive notes, CCs, aftertouch messages, all that is necessary in the firmware is to register the appropriate callback function. The MIDI stack is represented by an object called **Midi** for the first MIDI input (the leftmost MIDI connector), and **Midi2** for the second MIDI input (the rightmost MIDI connector).

In this firmware, we are going to register callbacks for incoming CC messages, and update the value of the encoders appropriately. To do this, we use the function **Midi.setOnControlChangeCallback()** to register a MIDI callback. A MIDI callback takes an array of bytes as a parameter, which represents the received message.

```
RangeEncoder param1(0, 127, “P1”);
RangeEncoder param2(0, 127, “P2”);
EncoderPage page(&param1, &param2);

void onCCCallback(uint8_t *msg) {
	if (MIDI_VOICE_CHANNEL(msg[0]) == MidiUart.currentChannel) {
		if (msg[1] == 1) {
			param1.setValue(msg[2]);
		} else if (msg[1] == 2) {
			param2.setValue(msg[2]);
		}
	}
}

void setup() {
	Midi.setOnControlChangeCallback(onCCCallback);
	GUI.setPage(&page);
}

void loop() {
	GUI.updatePage();
	if (param1.hasChanged()) {
		MidiUart.sendCC(1, param1.getValue());
	}
	if (param2.hasChanged()) {
		MidiUart.sendCC(2, param2.getValue());
	}
	GUI.update();
}

void handleGui() {
}
```

As you can see, the **onCCCallback()** function is register as the callback for incoming CC messages on the first MIDI input. It first checks to see if the channel of the message is the same as the current channel, and then checks to see if the CC number corresponds to either **param1** or **param2**. Of course, for a more complicated MIDI controller, we would store the encoders in an array and use a loop to set the correct value.

A very simple MIDI filter

We can also use these functions for input and output to create nice MIDI filters. For example, assume we want to convert incoming notes to make them match the C-major scale. We can do the necessary calculations by using the function **scalePitch()** out of the MidiTools functionality.

```
#include <MidiTools.h>

void onNoteOnCallback(uint8_t *msg) {
       MidiUart.sendNoteOn(MIDI_VOICE_CHANNEL(msg[0]),
                           scalePitch(msg[1], 0, majorScale),
                           msg[2]);
}

void onNoteOffCallback(uint8_t *msg) {
       MidiUart.sendNoteOff(MIDI_VOICE_CHANNEL(msg[0]),
                            scalePitch(msg[1], 0, majorScale),
                            msg[2]);
}

void setup() {
       Midi.setOnNoteOnCallback(onNoteOnCallback);
       Midi.setOnNoteOffCallback(onNoteOffCallback);
}

void loop() {
       GUI.updatePage();
       GUI.update();
}

void handleGui() {
}
```

The possibilities are of course endless, as you can transpose tracks, filter out messages, add new messages, etc...

Sequencing on the MiniCommand

A very important feature of the MiniCommand is it's ability to generate a tight MIDI clock, and also to receive a MIDI clock and synchronize to it. At the moment, only the first input can be used to receive MIDI clock information, but this will soon be extended to allow the second MIDI input to be used as well. As the MiniCommand uses a clever internal construction called a phase-locked-loop to synchronize to a MIDI clock, it can also be used to kind of "smooth" out a shaky MIDI clock signal.

The MIDI sequencing environment is handled by an object called **MidiClock**. It can be used to generate an internal clock, or used to synchronize to an external clock on **MidiUart**. It can then be used to call a callback on every 16th note, allowing you to run a function at regular intervals, and for example produce rhythms or melodic patterns. At the moment the sequencing environment is very simple, supporting only 16th notes and no swing, and not allowing for a very easy way to trigger notes of different lengths, and geared toward techno, but we are working on a very cool and flexible sequencing environment. The reason for this is that the MiniCommand was really developed with the MachineDrum in mind, and the flexibility of the environment only came up later.

In this sketch, we are going to generate a very simple random melodic pattern. The user can set the length of the pattern using an encoder, and randomize the played notes by pressing a button. To do this we create an array **melodyNotes[]**

containing the pitches of the notes to be sent. The callback function loops over this array, constrained by the length set by the encoder **lengthEncoder**. The notes in the array are randomized by the function **randomizeNotes()**, which generates arbitrary melodies over a two-octave major scale. The **on16Callback()** function is called on every 16th note, and uses the global 16th count variable called **MidiClock.div16th_counter**, which counts the number of 16th note since the **MidiClock** was last started.

```
RangeEncoder lengthEncoder(1, 16, “LEN”);
uint8_t melodyNotes[16] = { 0 };

void randomizeNotes() {
	for (int i = 0; i < countof(melodyNotes); i++) {
		melodyNotes[i] = scalePitch(48 + random(24), 0, majorScale);
	}
}

uint8_t prevNote = 0;

void on16Callback() {
	Midi.sendNoteOff(prevNote);
	prevNote = melodyNotes[MidiClock.div16th_counter %
	                         lengthEncoder.getValue()];
	Midi.sendNoteOn(prevNote, 100);
}
```

The internal clock is very easy to setup. The **MidiClock.mode** variable sets if the clock is generated internally (by setting it to **MidiClock.INTERNAL**), or if it syncs to an external clock source, which needs to send START and STOP commands as well, by setting it to **MidiClock.EXTERNAL**. In the case of an internally generated clock, we need to set the tempo by hand by calling **MidiClock.setTempo()**. At the moment these values are not yet really exact, so setting it to 100 bpm is more akin to setting it to 105 bpm, this will be fixed very soon. We can enable the sending of the MIDI clock signals on the output of the MiniCommand by setting the variable **MidiClock.transmit** to true. This can be used to chain out the MIDI clock as well (as the MiniCommand has no THRU). Finally, we enable the 16th note callback by calling **MidiClock.setOn16Callback()**.

```
	MidiClock.mode = MidiClock.INTERNAL;
	MidiClock.setTempo(100);
	MidiClock.setOn16Callback(on16Callback);
	MidiClock.start();
```

Finally, here is the whole sketch, which randomizes the notes on a button press. As you can see, sequencing on the MiniCommand is very easy, and a future API will allow for a much easier scheduling of events like notes of a certain duration, swing, and events that are not on the tempo grid. To see more complicated examples of sequencing, take a look at the MDArpeggiator sketch or at the MDPitchEuclid sketch. The SupaTrigga reverse functionality of the MDWesenLivePatch also relies on the tempo synchronization.

```
#include <MidiTools.h>

RangeEncoder lengthEncoder(1, 16, "LEN");
uint8_t melodyNotes[16] = { 0 };

void randomizeNotes() {
	for (int i = 0; i < countof(melodyNotes); i++) {
		melodyNotes[i] = scalePitch(48 + random(24), 0, majorScale);
	}
}

uint8_t prevNote = 0;
void on16Callback() {
	MidiUart.sendNoteOff(prevNote);
	prevNote = melodyNotes[MidiClock.div16th_counter %
	                       lengthEncoder.getValue()];
	MidiUart.sendNoteOn(prevNote, 100);
}

EncoderPage page(&lengthEncoder);

void setup() {
	randomizeNotes();
	lengthEncoder.setValue(8);
	MidiClock.mode = MidiClock.INTERNAL;
	MidiClock.setTempo(100);
	MidiClock.setOn16Callback(on16Callback);
	MidiClock.start();
	GUI.setPage(&page);
}

void loop() {
	GUI.updatePage();
	GUI.update();
}

void handleGui() {
	if (BUTTON_PRESSED(Buttons.BUTTON1)) {
		randomizeNotes();
	}
}
```

Interfacing with the MachineDrum

The whole reason why the MiniCommand was built was to interface with the Elektron MachineDrum. From there, it has evolved to become a very versatile MIDI controller, but a big part of the MidiDuino library focuses directly on communicating with the Elektron MachineDrum.

Control the MachineDrum's parameters

The first patch that was run on the MiniCommand (back when it was not freely programmable at all) was a small sketch to control the delay effect parameters. This alone opened up a whole new range of possibilities on the MachineDrum. This example shows how to write a very simple encoder page that controls the delay. We create four encoders and an encoder page. When the encoders are moved, we simply send out a control parameter for the MD ECHO effect (which actually sends out sysex data to the MD) using the function **MD.setEchoParam()**. Similarly, we have functions for the other effects called **MD.setReverbParam()**, **MD.setEQParam()** and **MD.setCompressorParam()**.

```
#include <MD.h>

RangeEncoder flfEncoder(0, 127, "FLF");
RangeEncoder flwEncoder(0, 127, "FLW");
RangeEncoder fbEncoder(0, 127, "FB");
RangeEncoder levEncoder(0, 127, "LEV");
EncoderPage page(&flfEncoder, &flwEncoder, &fbEncoder, &levEncoder);

void setup() {
  GUI.setPage(&page);
}

void loop() {
  GUI.updatePage();
  if (flfEncoder.hasChanged())
    MD.setEchoParam(MD_ECHO_FLTF, flfEncoder.getValue());
  if (flwEncoder.hasChanged())
    MD.setEchoParam(MD_ECHO_FLTW, flwEncoder.getValue());
  if (fbEncoder.hasChanged())
    MD.setEchoParam(MD_ECHO_FB, fbEncoder.getValue());
  if (levEncoder.hasChanged())
    MD.setEchoParam(MD_ECHO_LEV, levEncoder.getValue());
  GUI.update();
}

void handleGui() {
}
```

Similarly, we can easily control the parameters of a specific track by sending out CCs using the function **MD.setTrackParam()**, and trigger a track using the **MD.triggerTrack()** function. For this, the channel triggers on the MachineDrum have to be set to the default parameters. The following sketch does a simple polyrhythmic sequencer and randomizes the filter of track 3 on every hit (actually track 3 in the sketch is track 4 on the MachineDrum, because indexes start at 0 in MidiDuino). Similarly, LFO parameters can be set for every track using the **MD.setLFOParam()** function. For this sketch to work, the MachineDrum has to send out MIDI clock and CONTROL messages to the MiniCommand, so you have to connect the MachineDrum's output to the MiniCommand's first input, and the

```
#include <MD.h>

void on16Callback() {
  if ((MidiClock.div16th_counter % 3) == 0) {
    MD.setTrackParam(3, MODEL_FLTF, random(30, 100));
    MD.triggerTrack(3);
  }
}

void setup() {
  MidiClock.mode = MidiClock.EXTERNAL;
  MidiClock.setOn16Callback(on16Callback);
  MidiClock.transmit = false;
  MidiClock.start();
}

void loop() {
}

void handleGui() {
}
```

MiniCommand's output to the MachineDrum's input.

Read in the current MachineDrum Kit

Another feature that is available is parsing the MachineDrum's currently loaded kit. This feature is based on parsing a lot of sysex messages, and was written before the MiniCommand got an additional 128 kB of RAM. In the future, all the MachineDrum's message will be correctly parsed (and not just the kit messages). To read in the current Kit of the MachineDrum, use the function **MDSysex.getCurrentKit()**. This function accepts a callback function as argument. Once the current kit has been correctly received, the callback function is called. From this moment on, it can be assumed that all the **MD.trackModels[]** and **MD.trackParams[][]** settings are correct. However, there is no way for the MiniCommand to notice if the kit has been changed on the MachineDrum, so that you either have to use a continuously polling loop (which will disturb MIDI communication), or reload this data by hand when the kit is changed on the MachineDrum.

Once the current kit on the MachineDrum has been loaded, the sketch on the MiniCommand has access to all the current loaded machines and their parameters, which are stored in the variables **MD.currentGlobal**, **MD.currentKit**, **MD.name[]** (the name of the current kit), **MD.baseChannel** (the current base MIDI channel of the MachineDrum), **MD.trackModels[]** (the machine types of the current kit), **MD.trackLevels[]** (the current levels) and **MD.trackParams[][]** (the machine parameters of the current kit). Not all informations are parsed for now, but will soon be in a future version of the MD library. The following sketch reads in the current kit and displays its name when the upper left button is pressed.

```
#include <MD.h>

void onCurrentKitCallback() {
  GUI.setLine(GUI.LINE2);
  GUI.put_string_fill(MD.name);
}

void setup() {
  GUI.setLine(GUI.LINE1);
  GUI.put_string_fill("PRESS BUTTON1");
}

void loop() {
  GUI.update();
}

void handleGui() {
  if (BUTTON_PRESSED(Buttons.BUTTON1)) {
    MDSysex.getCurrentKit(onCurrentKitCallback);
  }
}
```

Playing notes on the MachineDrum

Once the kit information has been loaded, as shown in the previous sketch, you can start to play melodic notes on the MachineDrum. A certain number of machines are able to play melodic notes, and the necessary mapping from “real” pitch to the PTCH parameter on the MachineDrum is stored in the MD library. You can query if a track supports this melodic mapping by using the function **MD.isMelodicTrack()**. If this is the case, you can use the function **MD.sendNoteOn()** that will play a note on the specified track, taking care of converting the pitch value and sending a trigger. To be able to use this functionality, the standard note to track mapping has to be set up on the MachineDrum in GLOBAL -> MIDI -> MAP EDITOR: C2 triggers 01-BD, D2 trigger 02-SD, etc... This will be changed in a future version of the library that will parse the complete GLOBAL data.

The following sketch will load the current kit when the upper left button is pressed, and send notes received on the second MIDI input to the track selected with the first encoder. First, we set up the current kit parsing like in the previous sketch. We also use an encoder to configure the track on which the notes are going to be sent. When this track encoder is changed, we check if the selected track supports pitch information, and display the machine name if this is the case. Finally, in the callback function that is called when a note is received on the second MIDI input, we check if the track supports melodic information, and send out a note with **MD.sendNoteOn()** if that is the case.

```
#include <MD.h>

void onCurrentKitCallback() {
  GUI.flash_strings_fill("LOADED KIT:", MD.name);
}

RangeEncoder trackEncoder(0, 15, "TRK");
EncoderPage page(&trackEncoder);

void onNoteOnCallback(uint8_t *msg) {
  uint8_t track = trackEncoder.getValue();
  if (MD.isMelodicTrack(track)) {
    MD.sendNoteOn(track, msg[1], msg[2]);
  }
}


void setup() {
  GUI.setPage(&page);
  Midi2.setOnNoteOnCallback(onNoteOnCallback);
}

void loop() {
  GUI.updatePage();
  if (trackEncoder.hasChanged()) {
    uint8_t track = trackEncoder.getValue();
    GUI.setLine(GUI.LINE1);
    if (MD.isMelodicTrack(track)) {
      GUI.put_p_string_at_fill(4, getMachineName(MD.trackModels[track]));
    } else {
      GUI.put_string_at_fill(4, "XXX");
    }
  }
  GUI.update();
}

void handleGui() {
  if (BUTTON_PRESSED(Buttons.BUTTON1)) {
    MDSysex.getCurrentKit(onCurrentKitCallback);
  }
}
```

Further steps with MidiDuino

This was a very short overview of what is possible with the MidiDuino library, and you are encouraged to take a look at the provided firmwares. These are quite complex firmwares, with a lot more software engineering going on, so don't be afraid by their complexity. A good way to get into the development is to take the tutorial sketches, and start to modify them to control other parameters, to add your own callback functions and modification of the MIDI stream.

Also, this is an early release version of MidiDuino, and a lot of the planned functionality hasn't been implemented yet. The focus for the next release will be to include all the available additional resources that aren't leveraged yet (additional RAM, big flash storage) to be able to build crazy interesting firmwares. Look out for an update release! Also be sure to check the blogs on http://ruinwesen.com/ and our twitter feed at http://twitter.com/wesen for additional tutorials and firmwares.

Mididuino API reference

Main Functions

- void **setup**() { }
  - called on sketch startup
- void **loop**() { }
  - called regularly in a loop, main sketch function
- void **handleGui**() { }
  - called during the interrupt to process button switches, don’t put long running code in here, needs to be as fast as possible

GUI

Main Functions

- **GUI.update**()
  - update the display, call at the end of loop()
- **GUI.updatePage**()
  - update the current page, call at the beginning of loop()

Display Functions

- **GUI.setLine**(**GUI.LINE1**), **setLine**(**GUI.LINE2**)
  - set the current line of the display functions
- **GUI.put_value**(uint8_t idx, uint8_t value)
  - display a 8-bit number
- **GUI.put_value16**(uint8_t idx, uint16_t value)
  - display a 16-bit number
- **GUI.put_valuex**(uint8_t idx, uint8_t value)
  - display a 8-bit number in hexadecimal
- **GUI.put_value_at**(uint8_t idx, uint8_t value), **GUI.put_value16_at**(uint8_t idx, uint16_t value), **GUI.put_valuex_at**(uint8_t idx, uint8_t value)
  - display a number at a precise place

- **GUI.put_string**(uint8_t idx, char *str), **GUI.put_p_string**(uint8_t idx, PGM_P str), **GUI.put_string**(char *str), **GUI.put_p_string**(PGM_P str)
    - display a string (in RAM or in FLASH) on the display
- **GUI.put_string_fill**(char *str), **GUI.put_p_string_fill**(PGM_P str)
    - display a string and fill with whitespace
- **GUI.put_string_at**(uint8_t idx, char *str), **GUI.put_p_string_at**(uint8_t idx, PGM_P str), **GUI.put_string_at_fill**(uint8_t idx, char *str), **GUI.put_p_string_at_fill**(uint8_t idx, PGM_P str)
    - display a string at a specific place
- **GUI.flash_put_value**(uint8_t idx, uint8_t value), **GUI.flash_put_value16**(uint8_t idx, uint16_t value), **GUI.flash_put_valuex**(uint8_t idx, uint8_t value), **GUI.flash_put_value_at**(uint8_t idx, uint8_t value), **GUI.flash_put_value16_at**(uint8_t idx, uint16_t value), **GUI.flash_put_valuex_at**(uint8_t idx, uint8_t value), **GUI.flash_string**(char *str), **GUI.flash_p_string**(PGM_P str), **GUI.flash_string_fill**(char *str), **GUI.flash_p_string_fill**(PGM_P str), **GUI.flash_string_at**(uint8_t idx, char *str), **GUI.flash_p_string_at**(uint8_t idx, PGM_P str), **GUI.flash_string_at_fill**(uint8_t idx, char *str), **GUI.flash_p_string_at_fill**(uint8_t idx, PGM_P str)
    - display numbers and strings for a short duration
- **GUI.flash_strings_fill**(char *line1, char *line2), **GUI.flash_p_strings_fill**(PGM_P line1, PGM_P line2)
    - display 2 strings for a short duration

Buttons

- **Buttons.BUTTON1**, **Buttons.BUTTON2**, **Buttons.BUTTON3**, **Buttons.BUTTON4**, **Buttons.ENCODER1**, **Buttons.ENCODER2**, **Buttons.ENCODER3**, **Buttons.ENCODER4**
    - button and encoder number definitions
- **BUTTON_DOWN**(button), **BUTTON_UP**(down)
    - check if button is pressed down or released up, can be used in **loop**() as well
- **BUTTON_PRESSED**(button), **BUTTON_RELEASED**(button)
    - check if button was pressed or released, use only in **handleGui**()
- **BUTTON_DOUBLE_CLICKED**(button), **BUTTON_LONG_CLICKED**(button), **BUTTON_CLICKED**(button)
    - specialized functions for checking buttons

Encoders

- **RangeEncoder**(uint8_t min, uint8_t max, char *name)

- define an encoder with limits

- encoder.**hasChanged**()

  - check if an encoder has changed, call between **GUI.updatePage**() and **GUI.update**()

- uint8_t encoder.**getValue**()

  - get the current value of an encoder

- encoder.**setValue**(uint8_t value)

  - set the value of an encoder

- **EncoderPage**(**Encoder** *enc1, **Encoder** *enc2, ...)

  - define an encoder page (4 encoders with names)

- **GUI.setPage**(**Page** *page)

  - set the currently displayed page

MIDI Functions

Sending Messages

- **MidiUart.sendNoteOn**(uint8_t channel, uint8_t note, uint8_t velocity), **MidiUart.sendNoteOn**(uint8_t note, uint8_t velocity)

  - send a note on message

- **MidiUart.sendNoteOff**(uint8_t channel, uint8_t note), **MidiUart.sendNoteOff**(uint8_t note)

  - send a note off message

- **MidiUart.sendCC**(uint8_t channel, uint8_t cc, uint8_t value), **MidiUart.sendCC**(uint8_t cc, uint8_t value)

  - send a controller change message

- **MidiUart.sendProgramChange**(uint8_t channel, uint8_t program), **MidiUart.sendProgramChange**(uint8_t program)

  - send a program change message

- **MidiUart.sendPolyKeyPressure**(uint8_t channel, uint8_t note, uint8_t pressure), **MidiUart.sendPolyKeyPressure**(uint8_t note, uint8_t pressure)

  - send a polyphonic key pressure message

- **MidiUart.sendChannelPressure**(uint8_t channel, uint8_t pressure), **MidiUart.sendChannelPressure**(uint8_t pressure)

- send a channel pressure message
- **MidiUart.sendPitchBend**(uint8_t channel, int16_t pitchbend), **MidiUart.sendPitchBend**(int16_t pitchbend)
  - send a pitch bend message
- **MidiUart.sendNRPN**(uint8_t channel, uint16_t parameter, uint8_t value), **MidiUart.sendCC**(uint16_t parameter, uint8_t value)
  - send a NRPN message
- **MidiUart.sendRPN**(uint8_t channel, uint16_t parameter, uint8_t value), **MidiUart.sendCC**(uint16_t parameter, uint8_t value)
  - send a RPN message
- **MidiUart.sendRaw**(uint8_t *msg, uint8_t cnt), **MidiUart.sendRaw**(uint8_t byte)
  - send raw bytes
- **MidiUart.currentChannel**
  - Stores the default MIDI channel. Assign a new value to this variable to change the default channel.

Receiving Messages

- void **midiCallback**(uint8_t *msg)
  - prototype for a MIDI callback function
- **Midi.setOnNoteOnCallback**(midi_callback_t callback)
  - set a callback for receiving note on messages (check for velocity == 0 -> note off message)
- **Midi.setOnNoteOffCallback**(midi_callback_t callback)
  - set a callback for receiving note off messages
- **Midi.setOnControlChangeCallback**(midi_callback_t callback)
  - set a callback for receiving controller change messages
- **Midi.setOnAfterTouchCallback**(midi_callback_t callback)
  - set a callback for receiving aftertouch messages
- **Midi.setOnChannelPressureCallback**(midi_callback_t callback)
  - set a callback for receiving channel pressure messages
- **Midi.setOnProgramChangeCallback**(midi_callback_t callback)
  - set a callback for receiving program change messages
- **Midi.setOnPitchWheelCallback**(midi_callback_t callback)

- set a callback for receiving pitchwheel messages

Midi Clock and Midi Synchronization

- **MidiClock.mode** = **MidiClock.EXTERNAL**
  - set synchronization to external sync
- **MidiClock.mode** = **MidiClock.INTERNAL**
  - set synchronization to internal sync
- **MidiClock.transmit** = true / false
  - activate sending synchronizaton on MidiUart
- **MidiClock.start**() / **MidiClock.stop**() / **MidiClock.pause**()
  - control the clock engine
- **MidiClock.setTempo**(uint16_t tempo)
  - set the tempo when using internal synchronization (not correctly mapped yet)
- uint16_t **MidiClock.getTempo**()
  - get the tempo of the midi synchronization (a bit flaky)
- **MidiClock.setOn16Callback**(void (*callback)())
  - set a function to be called on each 16th note (in interrupt, so keep callback short)

Sequencing

- **DrumTrack**(uint32_t pattern, uint8_t len = 16, uint8_t offset = 0)
  - create a drum pattern
- bool drumTrack.**isHit**(uint8_t pos)
  - check if there is a hit at a certain position in the pattern
- **PitchTrack**(DrumTrack *track, uint8_t len)
  - create a pitch track linked to a drum pattern
- pitchTrack.**pitches**[]
  - array storing the note pitches of the pitch track
- pitchTrack.**playHit**(uint8_t pos)

  - play the pitch at a certain position (if there is a hit)

- **EuclidDrumTrack**(uint8_t pulses, uint8_t len, uint8_t offset = 0)
  - euclidean drum track sequencer
- **euclidDrumTrack.setEuclid**(uint8_t pulses, uint8_t len, uint8_t offset = 0)
  - reset the euclidean drum track parameters

Machinedrum API

- **MD.currentGlobal**
  - stores the number of the selected global on the MD, only up to date after a call to **MDSysex.getCurrentKit()**
- **MD.currentKit**
  - stores the number of the currently loaded kit, only up to date after a call to **MDSysex.getCurrentKit()**
- **MD.baseChannel**
  - sets the base channel of the MD, is updated by a call to **MDSysex.getCurrentKit()**
- **MD.trackModels[16]**
  - stores the track models of the MD, updated by a call to **MDSysex.getCurrentKit()**
- **MD.trackParams[16][24]**
  - stores the track parameters of the MD, updated by a call to **MDSysex.getCurrentKit()**
- **MD.trackLevels[16]**
  - stores the track levels of the MD, updated by a call to **MDSysex.getCurrentKit()**
- **MD.name[16]**
  - stores the name of the current kit, updated by a call to **MDSysex.getCurrentKit()**
- **MD.parseCC**(uint8_t channel, uint8_t cc, uint8_t *track, uint8_t *param)
  - interprets a CC and returns the track and the number of the original machinedrum parameter
- **MD.triggerTrack**(uint8_t track, uint8_t velocity)
  - trigger a track on the machinedrum with the given velocity. The machinedrum needs to be configured with the standard track MIDI note settings at the moment
- **MD.setTrackParam**(uint8_t track, uint8_t param, uint8_t value)
  - set the value of a machinedrum parameter

- **MD.setEchoParam**(uint8_t param, uint8_t value)
  - set a parameter of the global delay effect
- **MD.setReverbParam**(uint8_t param, uint8_t value)
  - set a parameter of the global reverb effect
- **MD.setEQParam**(uint8_t param, uint8_t value)
  - set a parameter of the global eq effect
- **MD.setCompressorParam**(uint8_t param, uint8_t value)
  - set a parameter of the global compressor effect
- **MD.trackGetPitch**(uint8_t track, uint8_t pitch)
  - get the converted PTCH parameter for a note of given pitch, the **MD.trackModels[]** have to be set correctly
- **MD.trackGetCCPitch**(uint8_t track, uint8_t cc, int8_t *offset = NULL)
  - gives back the pitch of received PTCH cc, the **MD.trackModels[]** have to be set correctly
- **MD.sendNoteOn**(uint8_t track, uint8_t pitch, uint8_t velocity)
  - send a note of given pitch to the melodic track, **MD.trackModels[]** have to be set correctly. Also, the track triggers for the MD have to be set to factory default
- **MD.sliceTrack32**(uint8_t track, uint8_t from, uint8_t to, bool correct = true)
  - if the given track has either a ROM-30, ROM-31 or RAM-P* machine loaded, the sample is assumed to be 2 bars long and sliced at the given 16th notes boundaries
- **MD.sliceTrack16**(uint8_t track, uint8_t from, uint8_t to)
  - if the given track has either a ROM-30, ROM-31 or RAM-P* machine loaded, the sample is assumed to be 2 bars long and sliced at the given 16th notes boundaries
- **MD.isMelodicTrack**(uint8_t track)
  - when **MD.trackModels[]** are up to date, returns if the track can be played as a melodic machine using **MD.sendNoteOn()**
- **MD.setLFOParam**(uint8_t track, uint8_t param, uint8_t value)
  - set the lfo parameter on the given track
- **MD.assignMachine**(uint8_t track, uint8_t model)
  - assign a given machine model to the track
- **MD.muteTrack**(uint8_t track, bool mute = true)

    - mute the track
- **MD.unmuteTrack**(uint8_t track)
    - unmute the track
- **MD.loadGlobal**(uint8_t id)
    - load the given global on the machinedrum
- **MD.loadKit**(uint8_t kit)
    - load the given kit on the machinedrum
- **MD.loadPattern**(uint8_t pattern)
    - load the given pattern on the machinedrum
- **MD.loadSong**(uint8_t song)
    - load the given song on the machinedrum
- **MD.setSequencerMode**(uint8_t mode)
    - set the sequencer mode (song or pattern) on the machinedrum
- **MD.setLockMode**(uint8_t mode)
    - set the song mode (classic or extended) on the machinedrum
- **MD.saveCurrentKit**(uint8_t pos)
    - save the current kit at the given position
- **MD.getMachineName**(uint8_t machine)
    - return a PGM_P string of the machine name for the given model
- **MD.getPatternName**(uint8_t pattern, char str[5])
    - store the pattern name in the given string ("A01", etc...)
- **MDSysex.getCurrentKit**(md_callback_t callback)
    - get the current kit by doing heavy sysex voodoo, and call the given callback function on completion