USB to WiFi+MIDI/RS232 Interface with ATXmega128A4U

 Status 28 February 2016

The circuit described here is primarily intended **to be used as a standard USB to MIDI interface as well as a standard USB to RS-232 interface**.

At full assembly, it is completed with a WiFi module (Microchip RN171XV) or an XBee module. In this case, **different modes of cross transfer between USB, WiFi and MIDI/RS-232 are available**. Especially MIDI may be controlled by OSC methods/commands.

The module is a simplified and more compact version of the "USB to WiFi+MIDI + DMX Interface" described at this website <www.midi-and-more.de/usbmiwidmx.htm>, **but without DMX option**. Most features else are the same.

It is **NOT allowed** to use this device together with any safety critical applications, where misfunction could result in personal injury oder noticeable material damage !

All information about this project is provided 'as is' – without any warranty nor responsibility !

Hardware

The USB interface is "full speed" grade, the device is **USB bus powered**. The supply current is less than 250mA. If no USB connectivity is needed or available, the USB connector may be used for regulated 5Volt power supply.

The microcontroller PLL multiplies the 6 MHz crystal up to 48 MHz as USB clock reference. From here the 24 Mhz CPU clock is derived, which is optimal for baud rate generation.

One USART output of the ATXmega processor feeds a 74HCT00 circuit, which inverts the signal and boosts the signal level from 3.3Volt to 5Volt to work as **MIDI OUT** as well as **RS-232 TxD** driver. The corresponding USART input of the ATXmega processor is connected open collector style with an optoisolated **MIDI IN** tied together with a transistor based **RS-232 input**. This kind of RS-232 circuitry looks primitive but has been proven to work very reliably.

Another USARTs of the ATXmega processor is used as serial interface for the WiFi/XBee module. Standard baud rates 9600 up to 230400 plus MIDI (31250 baud) are selectable.

A red/green dual LED and a blue LED signal presence of power and data flow.
While USB is connected, the dual LED lights amber in idle state - else it is red in idle state.
When data are received from USB, the green LED goes off for a moment, i.e.the light is red.
When data are received from MIDI / RS-232, the red LED goes off, i.e. the light gets green
When data are received from WiFi, the blue LED flashes, in idle state it is off..

Setting different modes of operation:

Different modes of operation are selected with 2 jumpers (or switches) J1 and J2.

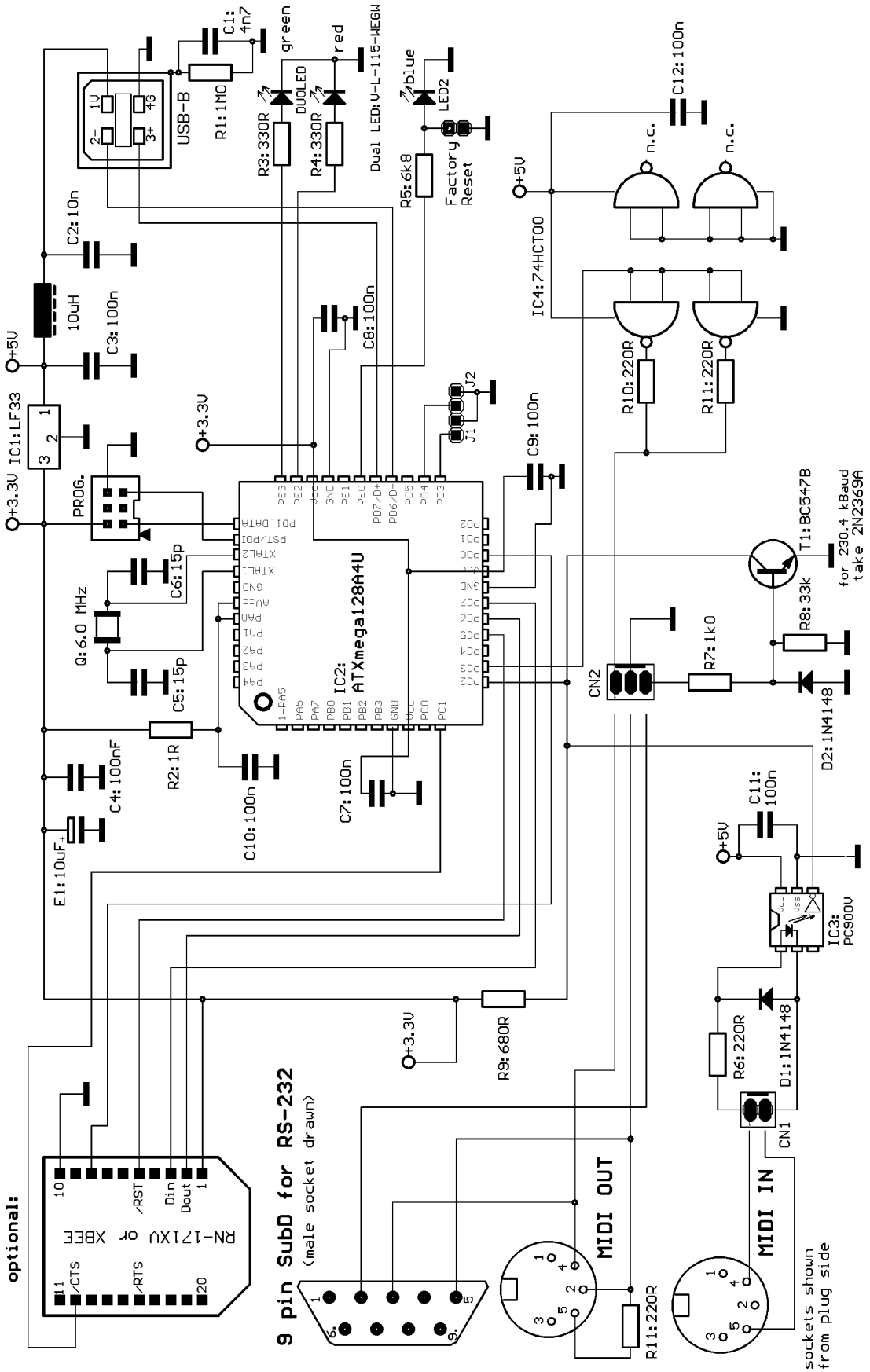
Jumper J1 selects which USB class is used for communication:

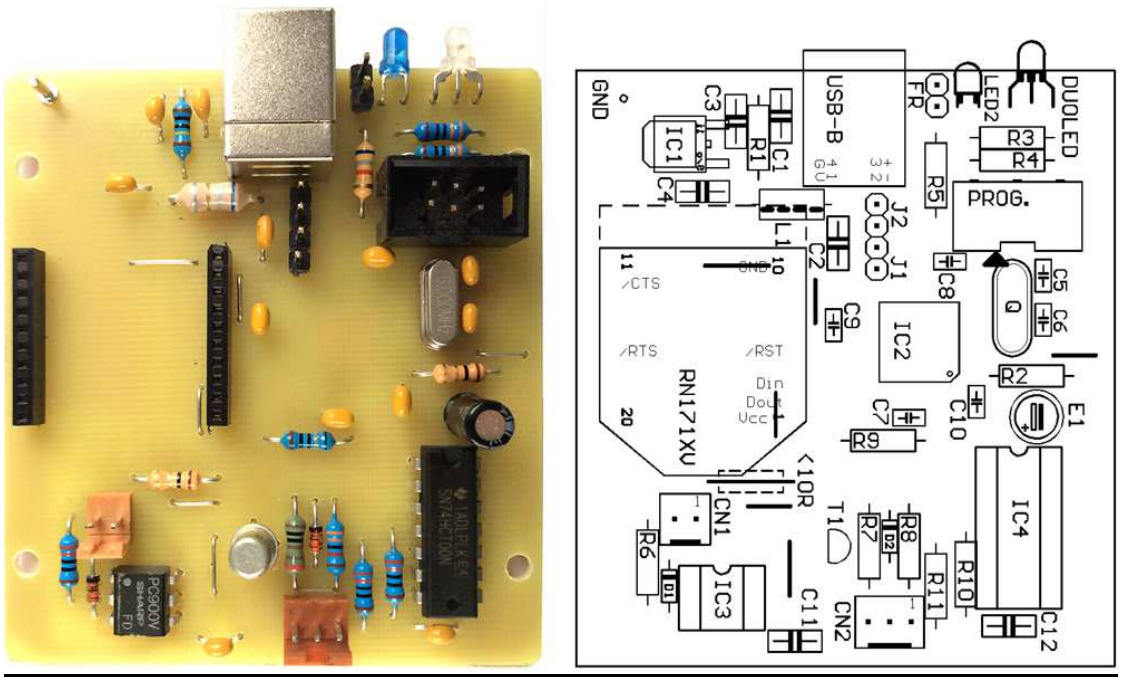
J1 not present or off: USB Communication Device (virtual COM port at the host)
J1 placed: MIDIStreaming device (virtual MIDI port at the host).

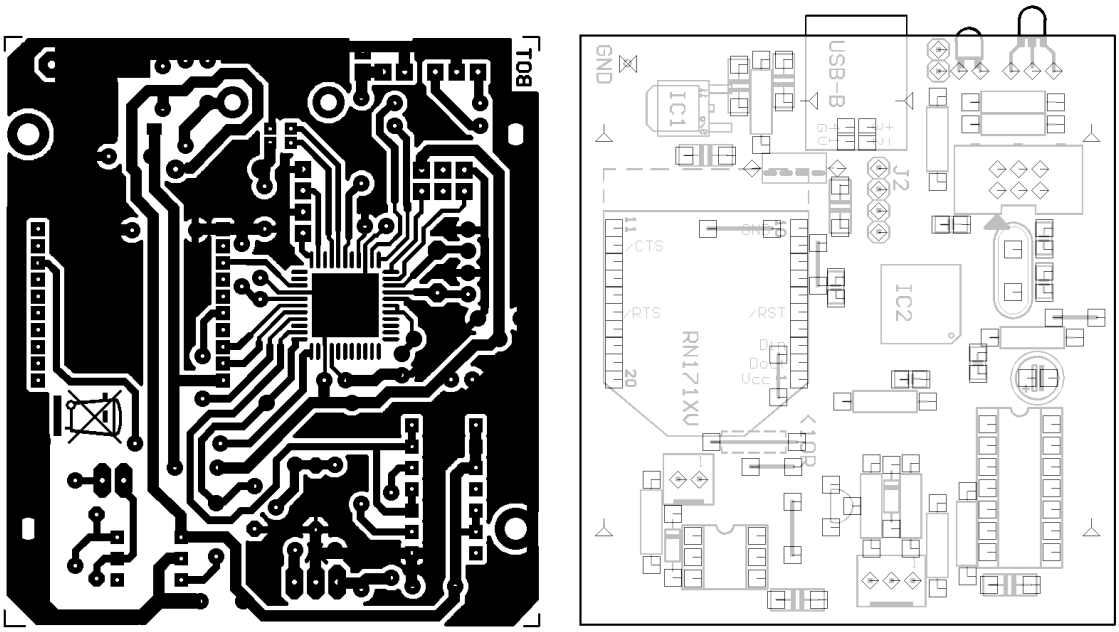
Jumper J2 enables some special modes of operation:

J2 not present or off: "triangular" 1:1 communication between all communication channels, as there is USB, MIDI/RS-232 and Wifi (or XBEE). The active USB class is determined by jumper J1.
J2 placed but J1 off: USB is in ASCII text based configuration mode (virtual RS-232 port). In parallel, Wifi communicates with MIDI/RS-232 exchanging OSC messages.
J2 and J1 both placed: USB is in MIDI mode. Wifi communicates with MIDI/RS-232 as well as with USB exchanging OSC messages. USB does **not** communicate with MIDI/RS-232.

Schematic:


USB-B
R1: 1MO
C1: 4n7
green
red
DUOLED
Dual LED:V-L-115-WEGW
R3: 330R
R4: 330R
blue
LED2
R5: 6k8
Factory Reset
C2: 10n
10uH
C3: 100n
+5V
IC1: LF33
+3.3V
PROG.
C8: 100n
J2
J1
C9: 100n
IC4: 74HCT00
n.c.
C12: 100n
R10: 220R
R11: 220R
T1: BC547B
for 230.4 kBaud take 2N2369A
R8: 33k
R7: 1kO
CN2
D2: 1N4148
C6: 15p
Q: 6.0 MHz
C5: 15p
IC2: ATXmega128A4U
R2: 1R
C4: 100nF
E1: 10uF
C10: 100n
C7: 100n
C11: 100n
IC3: PC900V
R9: 680R
R6: 220R
D1: 1N4148
CN1
optional:
RN-171XU or XBEE
/RST
Din
Dout
/CTS
/RTS
9 pin SubD for RS-232
(male socket drawn)
MIDI OUT
MIDI IN
R11: 220R
sockets shown from plug side





Drill symbols: 0.8mm 1.0mm 1.3mm 2.0mm 3.0mm

The PCB is designed to fit into a "Bopla Unimas85" enclosure, source Reichelt (BOPLA U 85) or Conrad 540803
It also fits acceptably well into an "Eurobox" enclosure, source Reichelt (EUROBOX) or Conrad part no. 523132
All resistors are carbon or metal film types, min 0.25W, max 5% tolerance.
All Capacitors should be multilayer ceramic 5.08mm respectively 2.54 mm raster, 50V
Microcontroller ATXmega128A4U-AU, TQFP44 case. Source RS Components.
Optocoupler: PC900V. Source Conrad (part no 184098)
Transistor T1: BC547B up to 115200 baud. For 230400 baud 2N2369A or 2N708 is recommended.
DUOLED: V-L-115-WEGW. Source Conrad (part no 187496).
Other LED types may be used, but then values of R3 and R4 have to be adjusted.
LED2: low current blue. $U_f$max (@0.1mA) = 3.0 Volt
Maximum value of R5 is 6.8 Kiloohm, else "Factory Reset" might not work properly.
CN1: source Reichelt (PS 25/2G BR) or Conrad. Wires may be soldered directly instead
CN2: source Reichelt (PS 25/3G BR) or Conrad. Wires may be soldered directly instead

Installation and configuration of the assembled module

The actual mode of operation is determined by the setting of jumpers J1 and J2.
To be operated from a front panel, the jumper pinheads should be connected with two toggle switches or an appropriate 4 position rotary switch.
The presence of jumpers is checked about every second by the firmware.

No jumper:

USB is operating in CDC/ACM mode (virtual COM port at the host)
Every byte received via USB is retransmitted 1:1 at MIDM/COM and WiFi
Every byte received via MIDI/COM is retransmitted 1:1 at USB and WiFi
Every byte received via WiFi is retransmitted 1:1 at MIDM/COM and USB

Only J1 placed:

USB is operating in MIDIStream mode (virtual MIDI port at the host)
Every MIDI message received at USB is retransmitted 1:1 at MIDM/COM and WiFi
Every MIDI message received via MIDI/COM is retransmitted 1:1 at USB and WiFi
Every MDI message received at WiFi is retransmitted 1:1 at MIDM/COM and USB

Only J2 placed:

ASCII text based onfiguration of system parameters and OSC method compiler using the USB port operating in CDC/ACM mode (virtual COM port at the host)
WiFi is operating as OSC (Open Sound Control) interpreter:
The content of MIDI related OSC data packets is retransmitted as MIDI messages from MIDI/COM.
MIDI messages received at MIDI/COM are sent via WiFi as OSC method formatted packets

Both, J1 and J2 are placed:

USB is operating in MIDIStream mode (virtual MIDI port at the host)
WiFi is operating as OSC (Open Sound Control) interpreter:
The content of MIDI related OSC data packets is retransmitted as MIDI messages from MIDI/COM and USB. Details see below.
MIDI messages received at MIDI/COM are retransmitted 1:1 at USB plus are sent via WiFi as OSC method formatted packets
MIDI messages received at USB are retransmitted 1:1 at MIDI/COM plus are sent via WiFi as OSC method formatted packets

In correspondence with the USB MIDIStreaming Class protocol, every MIDI data transfer is performed then in blocks of 1 – 3 MIDI bytes. This is handled automatically by the ATXmega firmware.

Change of jumper J1 changes the active USB communication class. When a change of J1 is detected during operation, the interface performs a complete reset including USB re-enumeration. Most times USB related software runnig on the host PC (at least under Windows) has to be restarted then.

Change of jumper J2 does not touch the active USB communication class. The interface performs a warmstart then without re-enumeration. Running PC software is not influenced.

When you first start the new programmed module as USB CDC/ACM device, the DUOLED initially glows dark red and it is shown in the Windows Device Manager as "Unkown Device". To be used as USB communication device (CDC/ACM class) with Windows, **additionally the appropriate "USB-MiWiCom.inf" file has to be installed**, which is available for download

from this website. Installation is done best the following way: Download this file. Select "Update Driver" in the Device Manager. Update/install "USB-MiWiCom.inf" manually.

When the microcontroller is programmed new, the default USB Vid is "6666". When operated as USB CDC/ACM device, the default USB Pid is "AFFA". In USB MIDIStreaming mode, the default Pid is "AFFB". **Please note that these experimental values are allowed for device test and evaluation only!**

For any use else, you have to enter your own Vid and Pid as follows:

Place only jumper J2. Connect the interface with an USB port of a PC. Start a terminal program on the PC with the appropriate virtual COM port. No handshake, baud rate does not matter. Enter exactly the sequence: `V yourVid` (as a sequence of 4 hex digits).

When this is finished, the letter `P` is promped.

Then enter `yourPid` (as a sequence of 4 hex digits) `<return>`

(Case independent. The hex words are entered without type specifier like 0x. Leading zeroes of the Vid/Pid MUST be entered, no spaces between the items). The input will be echoed. The MIDI Pid is always one higher. The new Vid/Pid becomes active during the next power cycle.

For your own Vid/Pid modify the .inf file correspondingly with an ASCII text editor.

Configuration of the USB / WiFi / MIDI interface board

Place only jumper J2. Connect the interface with a USB port of your PC. Start a terminal program on your PC with the appropriate virtual COM port. No handshake, baud rate does not matter.The following parameters are stored permanently and recalled during every power cycle or reset of the interface. Following items may be configured:

Baudrate at MIDI + RS232 I/O: type `B <baudrate> <return>`

Following baudrate values are accepted (**only the first two digits have to be entered**)

9600 type 96	19200 type 19	MIDI (31250) type 31	
38400 type 38	57600 type 57	115200 type 11	230400 type 23

Alternatively the baud rate can be adjusted semi-automatically while the interface is powered up: Send a series of ASCII characters ('m' and 'U' work best) immediately within 1 sec after power on or reset. If recognized, the new baud rate is stored permanently to be loaded at any power cycle

Baudrate for RN-171 or XBee: type `W <baudrate> <return>`

Baud rate parameters are the same as for MIDI/RS-232 I/O. No semi-automatic baud rate detection.

OSC path filter: type `P <one decimal digit 0…9 > <return>`

The meaning of this parameter is explained in the OSC chapter below

OSC MIDI input handling: type `M <one decimal digit 0…3 > <return>`

The meaning of this parameter is explained in the OSC chapter below

USB Vid/Pid:see above

Check the actual configuration: type ?

Additionally, you can **enter and check user defined OSC methods**, details see p.15-19.

Emergency Reset configuration to state of delivery

Power OFF, place a jumper on the pinhead near the blue LED, cycle power. The microcontroller of the interface board **exclusively resets communication relevant items and resets the USB Vid/Pid to default values**, no user defined OSC methods / strings are changed. **The RN171 module performs a hardware controlled "factory RESET" as described in its user manual**, !! takes about 7 sec. After reset, the device remains in an endless loop. Remove the jumper, cycle power.

Installation and configuration of the RN171XV module

Before practical use the microcontroller of the WiFi / MIDI interface as well as the RN171 module have to be configured. **Use and configuration of XBee** modules see page 8.

Whereas configuration of the USB /WiFi / MIDI interface board itself is simple, configuration of the RN171XV module takes some care and reading of its user manual. It is not inteded here to copy that manual, only the most inportant hints can be given here to make it operating best for the USB /WiFi / MIDI interface board.

Configuration of the RN171XV module

First **install the latest firmware of the RN171XV module** (version 4.75 when this is written) as described in the RN171 user manual. For good operation, it is necessary to use a firmware version which supports "AP mode" instead of "Adhoc mode".
It is a good idea, first to look at the Microchip website for actual information about versions and handling instructions (see www.microchip.com/wireless/firmware and /FAQs)

For firmware download you have to **put the the RN171 module in "command mode"**. Though there may exist alternative methods, the most convenient approach is to use the RS-232 port or USB.

For this reason connect your PC with USB or use the optional 9 pin subD connector.
Start a terminal software ("Teraterm" recommended for Windows, "CoolTerm" for Mac OS) with the appropriate COM port and baud rate setting.

Two different baud rates have to be taken into account:

First the **baud rate between the interface board and the RN171XV module**. The default setting with new programmed firmware is 9600 baud (both the interface microcontroller and the RN171XV module). Don't change it before all configuration is done. Later, for practical use, 230400 baud is recommended.

If RS-232 is used for firmware update (USB is baud rate independent): **The RS-232 port baud rate of the MIDI/COM interface** can be set manually in configuration mode (details see above p.5) or semi-automatically during power on. For semi-automatic baud rate selection type a series of letters ('m' and 'U' work best) while the module is booting up. If baud rate detection is successful, an appropriate text is responded to the terminal with the correct baud rate. Besides MIDI, baud rates 9600, 19200, 38400, 57600, 115200 and 230400 are supported.

The following hints are based on best knowledge but given without any warranty. You are operating at your own risk.

To get the RN171 module into command mode, use 1:1 RS-232 mode communication (no jumpers placed). **Send the byte sequence "$$$"** from the terminal. If successful, the RN171 module will prompt with "CMD" . (Bytes are transferred 1:1 through the interface board because it is in "transparent" mode and translates between different baud rates). Now you are in the command (=configuration) mode of the RN171 module as described in its manual.

Check the actual RN171 firmware version by typing `ver <return>.` If your actual version is equal or higher than 4.0, update is not absolutely necessary but recommended.
Next check the RN171 file system (for later comparison) by typing `ls <return>`
For firmware update, it is essential that the Wifi communication of the RN171 module is set to "infrastructure mode" (usually default). Check this by typing `get w <return>.` In the returned output, check item "JOIN", the parameter should be 0 or 1.
If different, type `set w j 0 <return>`
Next type `get i <return>`, the item DHCP should be ON,
else adjust it by typing `set i d 1<return>`.

If for any reason you prefer to assign a fixed IP address to your interface (see `set ip address` , `set ip netmask, set ip gateway, … etc`), be sure that it is conformant with the overlaying network.

Now list available networks by typing `scan <return>`

Join to your local network by typing `set w s <your network SSID> <return>`
`set w p <your WPA passphrase> <return>`
`join <your network SSID> <return>`

If this is done successfully, an appropriate text message will be returned. Else read the manual for possibly necessary additional checks and settings.

If successful, download the actual firmware. To check the actual settings for this purpose, type:

`get d b <return>` The response should be `Backup=rn.microchip.com`

If not type `set d b rn.microchip.com <return>`

Next type `get f u <return>`

A list should appear which essentially contains the lines

```
FTP=0.0.0.0:21
File=wifly-EZX.img
User=roving
Pass=Pass123
```

If not, correct these items with the appropriate set commands.

If ok you can start the download with

`ftp update wifly7-441.img <return>` **(or newer version if available from Microchip)**

If your present firmware is 4.0 or higher, you can update a .mif file, which contains the image plus additional features (details see RN171 manual)

`ftp update wifly7-441.mif <return>` **(or newer version if available from Microchip)**

While the download process is going on, a row of dots will be displayed at your terminal window. Sometimes you have to try several times. **After the download has finished**, **it is absolutely necessary to restart the RN171 module** as described in the manual by the command sequence **`$$$ factory RESET reboot $$$`** before going on.

Check the actual RN171 version and file system by typing `ver <return> ls <return>`

The next step is to **prepare the module for appropriate communication**.

This depends essentially on the way you want to use the interface in practice, so no precise "to do list" can be offered here. Only the most important stumble stones will be mentioned.

A basic decision is if you want to run the interface in **Infrastructure mode** (as a member of an existing network with external WiFi access point) or in **AP mode** (the RN171 module works as independent access point and spans its own network). This offers some advantage in combination with handheld mobile devices in spontaneous situations and with minimal overhead. Differing from the similar "USB to WiFi+MIDI + DMX Interface" this setup can be only made by text input in the RN171 command mode.

To get associated to an Infrastructure network is described above.

AP mode is activated by software with the command `set w j 7 <return>`. The RN171 module creates an open network with the fixed name "WiFly-EZX-xx". This will be visible at WiFi counterparts a few seconds after the WiFi / MIDI interface was powered up or reset. The RN171 module always uses the IP address 192.168.1.1 (and default port 2000 for Telnet sessions). IP addresses are assigned to the communication partners by DHCP - first partner usually (not guaranteed) 192.168.1.11.

In AP mode it's your task to configure the communication partners compatible with the network created and managed by the RN171 module. Be sure that parameter DHCP=SERVER, else enter `set i d 4 <return>`. Most times the data latency in AP mode is shorter than in infrastructure mode.

To configure the module for **TCP/IP** communication (Telnet), type `set i p 2 <return>`

For MIDI communication with existing Ethernet capable counterparts, in most cases **UDP** is appropriate. If you prefer UDP communication, type `set i p 1 <return>`. **For UDP**, in addition to the modules own IP address, **3 important parameters have to be configured**:

--- the host IP address (where UDP data will be sent to) by `set ip host`
you may use a broadcast IP address (in AP mode 192.168.1.255) for easier access to unkonwn IPs.
--- the remote port (where UDP data will be sent to) by `set ip remote`
--- the local port (on which this device will receive data) by `set ip localport`
Commonly used port numbers for MIDI capable applications in UDP mode are 8000, 8080, 9000

Note, that now for **TCP sessions, the port name is the same as the UDP local port !**

Attention has to be paid to the way how the RN171 assembles bytes to WiFi sent packets. For **optimum correspondence with the timing of the interface firmware**, following parameter has to be adjusted `set c t 1 <return>` . For details see RN171 manual.

Last not least, though they are useful for debugging, switch off all system messages as far as possible after you have got the interface running your way - type `set s p 0 <return>`

Finally set the **baud rate for communication between the interface board and RN171** (differing from the baud rate at MIDI I/O !) to the highest possible speed 230400 baud. Type `set u b 230400 <return>`. Don't forget to adjust the baud rate of the interface microcontroller to the same value after you have finished the next two steps:
To store your settings permanently, type at the end of the session: `save <return>`
To make your setup active and leave configuration, type `reboot <return>`

Installation and configuration of an XBee module

If a simple "wireless MIDI cable" with minimal effort for setup and handling is needed, the use of XBee modules may be preferred to a WiFi solution. For a fast and easy point-to-point connection, the more simple 802.15.4 protocol modules should be used instead of those with built-in Zigbee network support. Simple multi-point broadcast is supported implicitly. For more complex arrangements and configuration details you are referred to the XBee manual.

If an XBee module shall be used instead of the RN171 module, **the USB / MIDI / WiFi interface board has to be operated in the WiFi AP mode (rotary switch in pos 0 to 7)**. "Factory reset" for the RN171 is not supported by XBee modules. Else the PCB design and the firmware is compatible for XBee modules as well as for RN171.

For a first functional test two XBee modules may be used out of the box without configuration.The default baud rate is 9600 - but it is recommended to set the UART to 115200 baud. 230400 baud is not supported by XBee.

To put the XBee module into configuration mode: set the rotary switch to pos.3, start a terminal program via RS-232 or USB. Type the sequence +++ and wait until "ok" is prompted. Send "atbd 7<return>" to set baud rate 115200. After "ok" is prompted, check setup with "atbd<return>". If the entry is correct send "atwr<return>" to store permanently. Go back to data mode by "atcn<return>". **Wrong entry can make the XBee module unusable!**

Additional note if one USB / MIDI / Wifi(=XBee) interface shall communicate with other types of XBee modules:
XBee transmission itself is baud rate independent and converts correctly between different endpoint UART baud rates. **This way the XBee connection is useable as a wireless baud rate converter.**

Command sequence "atbd 7a12<return>, sets the XBee UART to MIDI baud rate, which may be useful if the XBee module shall be configured for use in a **wireless converter for a sound card gameport MIDI interface (MPU-401)**.
For a corresponding DiY building instruction see "www.midi-and-more.de/xbee-mpu.htm"

OSC("Open Sound Control") methods for MIDI I/O (J2 or both jumpers placed)

All OSC capable software known here uses UDP for WiFi/Ethernet communication
A preferred OSC default port for received and transmitted datagrams is 8000. Sometimes 8080 or 9000 is more appropriate. This may be changed for differently configured networks in RN171 command mode.

Every command string (OSC slang "methods") of the subsequently described **predefined standard commands** esentially consists of 2 letters, which may be entered case insensitive. This restriction is not as strong as it may look:

Preceeding **path descriptions**, which contain a mixture of slashes '/' and decimal ciphers, are ignored by the default configuration (config parameter '[P' set to ASCII '0'). If a different decimal cipher is configured, the first character following the first slash in the received comand/method is taken as filter criterion: if it matches, the command passes, else it is ignored. When it passes, everything of the path is ignored until the next slash which is followed by a literal character (A to Z, case insensitive) appears. If any sequence like this is contained in your path description - sorry - this will be interpreted as method name or configure [P=0.

Apart from a preceeding path **every OSC command (=method)** string is definitively parsed and selected concerning the first 2 alphabetic letters (exception MULTI… , see below). All subsequent letters are omitted until the first ASCII representation of a decimal cipher (0...9) appears.

This "numerical" sequence is parsed as an integer number and temporarily stored as the "**item number**" of this command which is handled in some way like an OSC argument. This parsing procedure is finished when a null byte or the comma in the OSC command string appears. The rest is parsed in conformance with general OSC standards.

The **order of processing** is organized as follows: When a new OSC command/message is received and passed the simple path filter, **first the command/method name, followed by the item number and all OSC arguments are stored** in a temporary memory before anything is processed. Up to 8 arguments are allowed per command/method. "Temporary" means that these numerical values are stored and referenced as long as the actual command/method is processed. Will be overwritten by the next command/method.

Next the validity of the command/method name is parsed, checked and transferred to the routine which evaluates and executes the parameters according to the specific features of this command/method. If an OSC MIDI, STRING or BLOB argument is contained in the message, which has no correspondence to the respective command, it is ignored and the remaining arguments (i and f) are "consumed" in the order of appearance.
Integer arguments (type i) are always consumed and executed "as received". Values greater than (context given) maximum values are limited to the max. allowed value (in most cases 127 or 255) .Negative values are limited to zero.
Float arguments (type f) in the range 0.000 <= 1.000, are recalculated to an integer ranged from 0 to 127 (or the max. range allowed by the actual context). Float 1.000 is rounded down to the highest context conformant integer (in most cases to 127 or 255). If the float argument is greater than 1.000, it is rounded down to the next integer value. This latter kind of coding is possible but not recommended, because there is an ambiguousity when the argument is the range: greater than exactly 0.000 and less or equal 1.000.

Data to be transmitted from MIDI OUT and USB:

The OSC feature of this USB / WiFi / MIDI interface seems to be most useful in combination with handheld devices like iPad, iPhone or Android based ones. For users which won't build their personal interface, the patch "MIX2" included by default in TouchOSC and the default interfaces downloaded with "Control" seem to be most useable for the applications intended here and are used for reference as far as possible.

Every OSC method for MIDI transmission **works simultaneously on the native MIDI OUT as well as on USB**. Exceptions: rotary switch positions 3 and B, also see /TX and /TU.

> An inherent problem of this kind of signalling is that there is no generally useable communication channel to handle error messages easily. Consequently, faulty commands are handled as smooth as possible. Context oriented details are discussed below.

There are different levels of complexity (= freedom) of operation. This description starts from the most simple methods to more complex and general ones. As explained above, for every command/method only the first two significant letters are parsed. The rest is ignored until a decimal cipher (0…9) does appear in the string.

As a consequence, no decimal ciphers are allowed inside the method string itself!

Exceptionally any string beginning with "MULTI.. " is parsed up to the next letter following the "I". If it is an "F", the TouchOSC "Multifader" method is assumed and two subsequent numerical "item numbers" are parsed.

The first "item number" is the "group", the second one is the "member" in the group. Internally this two-dimensional array is recalculated into a single "item number" and then processed like a single FADER. The recalculation is done as follows: item number = (group-1)*MembersPerGroup + member. Default MembersPerGroup=16. The same way, a following "P" is handled as TouchOSC "Multipush" (default ColumnsPerRow=12) and a following "T" is handled as "Multitoggle" (default ColumnsPerRow=8). Columns and rows are changed here in contrast to normal "mathematical feeling".

If the letter following "I" is a "B", the method string is interpreted as method "Multibutton" supplied by "Control" with only one numerical "item number" processed the same way as PUSH, correspondingly a following "S" is interpreted as "Multislider" processed the same way as FADER.

The least complex and most common methods: (all letters case insensitive!!)

Any message (with any item number) received from TouchOSC or others like

/1/PUSH47 + argument (type i or f)

or shorted as /PU47 + argument (i or f) or short form of "BUTTON": /BU47 + argument (i or f)

> will be forwarded to MIDI OUT as NOTE ON, preset MIDI channel (see command /SC below, default channel 1), "item number" = Note Value (47 in the example). If the argument is an integer, it will be retransmitted as velocity of this note. If the argument is a float = 1.000, the velocity is sent corresponding to the actually stored value (see command /SV below, default = 127). If the argument is integer 0 or float 0.0000, it is always sent as NOTE ON, Note Value 47, velocity=0. Any float value in between (if possible with your personal PUSH version) will be scaled 0 to 127. This way different kinds of virtual MIDI keyboard may be installed easily on the mobile device.

/1/FADER7 + argument (type i or f)

or shorted as /FA7 + argument (i or f) or short form of "SLIDER": /SL7 + argument (i or f)

> will be forwarded to MIDI OUT as CONTROL CHANGE, preset MIDI channel (see command /SC below, default channel 1), "item number" = Controller Number (7 in the example), Controller Value as argument, internally scaled to 0-127 how ever possible – see above. This way different kinds of MIDI 7 bit controllers may be installed easily on the mobile device.

/1/TOGGLE3 + argument (type i or f) shorted as /TO3 + argument (type i or f)

> will be forwarded to MIDI OUT as PROGRAM CHANGE, preset MIDI channel (see command /SC below, default channel 1), "item number" = Program Number 3 (i.e. data byte = 2 in the MIDI message!) The PROGRAM CHANGE message will only be sent **if the argument is greater than 0.000**.

All commands (OSC "methods" including the "path") and OSC "Type Tag Strings" have to be filled up - if necessary - with postponed nullbytes to fit into a 32 bit raster accorcding to the OSC specification.

Non accepted or faulty coded commands are ignored without error message. Faultly arguments are rounded to max context conformant value.
Regarded from the perspective of practical operation, a sequence of UDP datagrams with OSC content is processed by the WiFi / MIDI interface as a continous stream.

Trigger more specific MIDI channel messages:

The names of the subsequently described methods are chosen result oriented but short and neutral – preferably to be used for user-written OSC method generators and to be used as substitutes and arguments in the simple OSC method compiler described below.

/NZn + (optional) argument1 =MIDI channel 1-16
triggers transmission of a NOTE OFF message. The Note Value (pitch) is given by n (0..127)
The optional argument1 inserts a particular MIDI channel (1..16) into the message. If no argument is received, the preset MIDI channel (see method /SC below) is inserted into the message (default=1)

/NOn + (optional) argument1 =velocity + (optional) argument2 =MIDI channel 1-16
triggers transmission of a NOTE ON message. The Note Value (pitch) is given by n (0..127).
The optional argument1 inserts a particular velocity (0..127) into the message. If missing, the default velocity is used (see method /SV below). The optional argument2 inserts a particular MIDI channel (1..16) into the message. If no argument2 is received, the preset MIDI channel (see method /SC below) is inserted into the message (default=1). If no arguments are received, the MIDI message is sent with the preset MIDI channel and velocity. Argument 2 cannot be used without preceeding argument 1

/NDn + (optional) argument1 =velocity + (optional) argument2 =MIDI channel 1-16
triggers transmission of a NOTE ON message to the General MIDI Drum Set. The Note Value (selects drum/percussion instrument) is given by n (0..127).
The optional argument1 inserts a particular velocity (0..127) into the message. The optional argument2 inserts a particular MIDI channel (1..16) into the message. If no argument2 is received, the preset MIDI Drum Set channel (see method /SD below) is inserted into the message (default=10). If no arguments (1 and 2) are received, the MIDI message is sent with the preset MIDI channel and velocity.

/CCn + argument1 = controller value+ (optional) argument2 =MIDI channel 1-16
triggers transmission of a CONTROL CHANGE message. The controller number is given by n (0..127).
Mandatory argument1 inserts the controller value (0..127) into the message. The optional argument2 inserts a particular MIDI channel (1..16) into the message. If no argument2 is received, the preset MIDI channel (see method /SC below) is inserted into the message (default=1). Then this command is equivalent to command FADER as described above.

/PRn + (optional) argument1 =MIDI channel 1-16
triggers transmission of a PROGRAM CHANGE message. The program mumber is given by n (0..127). **This is the data byte which gets inserted in the MIDI message!** Please note, that most MIDI software and other MIDI equipment formally use program numbers in the range 1 to 128, i.e. one higher as the one which is actually sent as MIDI data byte. **So enter the program number you find in manuals etc. decreased by 1 in the OSC message !**
The optional argument1 inserts a particular MIDI channel (1..16) into the message. If no argument is received, the preset MIDI channel (see method /SC below) is inserted into the message (default=1)

/POn + (optional) argument1 =velocity + (optional) argument2 =MIDI channel 1-16
triggers transmission of a POLY KEY PRESSURE (=POLYPHONIC AFTERTOUCH) message. The Note Value (pitch) is given by n (0..127).
The optional argument1 inserts a particular velocity (0..127) into the message. If missing, the default velocity is used (see method /SV below). The optional argument2 inserts a particular MIDI channel (1..16) into the message. If no argument2 is received, the preset MIDI channel (see method /SC below) is inserted into the message (default=1). If no arguments are received, the MIDI message is sent with the preset MIDI channel and velocity. Argument 2 cannot be used without preceeding argument 1

/PB +argument1 + (optional) argument2 = MIDI channel 1-16)
/PIT +argument1 + (optional) argument2 = MIDI channel 1-16) (**syntax exeption**, see "/ping", p.21)
both trigger transmission of a PITCH BEND message (7, 8 or 14 bit accuracy). Any numerical "item number" appended to the method name is ignored.
Mandatory argument1 inserts the pitch bend into the message.

If it is received as an integer argument, any value between 0 and 16383 (14 bit) is accepted, which is processed in conformance with the MIDI standard: the 7 most significant bits are transmitted as second MIDI data byte. All less significant bits (if the integer is greater than 127) are transmitted in the first MIDI data byte "7 bit left adjusted". Else the first MIDI data byte is transmitted as zero.
If it is received as a float argument, the range 0.000 to 1.000 is blown up to 14 valid integer bits, which are transmitted as first and second data byte of the MIDI PITCH BEND message as described above for integer parameters.
The optional argument2 inserts a particular MIDI channel (1..16) into the message. If no argument2 is received, the preset MIDI channel (see method /SC below) is inserted into the message (default=1).

/PCn + (optional) argument1 =MIDI channel 1-16
triggers transmission of a CHANNEL PRESSURE message. The message content is transported by means of the "item number" n (0..127).
The optional argument1 inserts a particular MIDI channel (1..16) into the message. If no argument is received, the preset MIDI channel (see method /SC below) is inserted into the message (default=1)

Enter specific numerical values to be inserted in MIDI messages:

To enter numerical values exactly on a mobile device without numerical feedback, some ROTARY widgets are reseved for a special function:
In the app they are round symbolic potentiometers, but simplified in the PD demo patcher as linear sliders.

Evaluation of ROTARY1,2,3,4,5, 6,127 and 255 gets "digitized" as follows:

Any ROTARY1,2,3 generates only 5 distinct numerical levels in the WiFi / MIDI interface, namely 0, 25%, 50%, 75% and 100% of full range. Separation of levels is put in the middle between, so selection of these 5 levels is done precisely without fiddeling around.
ROTARY1 generates values 0, 1, 2, 3, 4.
ROTARY2 generates values 0, 5, 10, 15, 20.
ROTARY3 generates values 0, 25, 50, 75, 100.
ROTARY4 generates values 0, 100, 200 with level 100 at more or less middle position
These 4 values are added up (accumulated) inside the interface every time another one of these commands/methods is reveived. This way a continous precise setting of numbers 0 to 255 is provided without numerical feedback. **Attention: the accumulated Rotary Sum is reset to 0 during power cycle or reset !**

As an alternative, with **ROTARY127 (or same way ROTARY5)**, which is especially useful for widgets with numerical feedback, the **full range 0 to 127** can be selected. Primarily a number is generated this way, it has to be assigned or copied to the related feature by the commands/methods which are described next. **For proper decoding, the argument of any ROTARY1 to 5 and 127 has to be a float in the range 0.000 to 1.000 or an integer in the range 0 to 127**

As another alternative ROTARY255 (or same way ROTARY6) primarily generates a number in the **full range 0 to 255**. It has to be assigned or copied to the related feature by the commands/methods which are described next. **For proper decoding, the argument of ROTARY6 or 255 has to be a float in the range 0.000 to 1.000 or an integer in the range 0 to 255.**

/SB (no argument)
transmit the accumulated Rotary Sum (limited to 255) immediately from MIDI OUT

/SC (no argument)
stores the accumulated Rotary Sum (1 to 16) as MIDI channel which is inserted into any subsequent MIDI message (if not overwritten individually by argument)

/SD (no argument)
stores the accumulated Rotary Sum as MIDI drum set channel to be inserted into subsequent /ND commands (if not overwritten individually by argument2)

/SV (no argument)
stores the accumulated Rotary Sum as veloctity to be inserted into subsequent /NO and /PUSH commands (if not overwritten individually by argument1)

/SF (no argument)
stores the accumulated Rotary Sum as "MembersPerGroup" to be applied at subsequent TouchOSC MULTIFADER commands (default = 16)

/ST (no argument)

stores the accumulated Rotary Sum as "RowsPerColumn" to be applied at subsequent TouchOSC MULTITOGGLE commands (default = 8)

/SP (no argument) provides more complex features:

Rotary Sum <100:stores the accumulated Rotary Sum as "ColumnsPerRow" to be applied at subsequent TouchOSC MULTIPUSH commands (default = 12)

Sum: >=100 but <200: subtacts 100 and **performs /SF** **Sum>=200**: subtacts 200 and **performs /ST**

Attention: The actual input of ROTARY1,2,3,4,5,6,127,255 is permanently added up and kept available until another ROTARY message is received. During reset or power cycle, the sum is cleared to zero and is not updated until any ROTARY message comes. So the sum has to be updated after every power cycle or system reset – or even better before actual use.

/SA (no argument)

stores the parameters entered by the commands/methods described above **permanently** to be used as default after next power cycle or system reset. **ROTARY values and the accumulated Rotary Sum are not stored permanently.**

Transmit System Exclusive and other messages from MIDI OUT and USB:

The method/command /TX is a kind of general container which triggers transmission of one or several bytes from MIDI OUT. The argument type is detected automatically and executed appropriately.

/TX +argument: = 0 .. 255 (OSC type tag "i") or 0.000 to 1.000 (OSC type tag "f")

Retransmits a single byte received as 32 bit Integer (OSC type tag "i")
I.e. only the least byte will be retransmitted 1:1 if the first 3 bytes are all zero. Greater 32 bit integer values are limited to 255. Float tags 0.000 <= 1.000 are expanded to the range 0 - 255

/TM +argument: = 0…127 (OSC type tag "i") or 0.000 to 1.000 (OSC type tag "f")

The float tags are expanded to the range 0 – 127, integers are limited to 127.
All tag types else (b,l,m,s) are processed the same way as by command /TX

/TX +argument: = null terminated string (OSC type tag "s")

Transmits an arbitrary string up tp 255 bytes length
To recognize the length of the string, it must be terminated by a null byte.
Furthermore the end of the string has to be filled up with additional null bytes to fit into the 32bit/=4byte raster according to the OSC specification. In PD patchers this is formatted automatically.

/TX +argument: = counted sequence of arbitrary bytes (OSC type tag "b")

Transmits an arbitrary data block of data up to 255 bytes lenth
The length of the data block is declared by a preponed count integer according to the OSC specification. Differing from a string, the data block may contain any byte values, even null bytes. But the end of the data block has to be filled up with additional null bytes to fit into the 32bit/=4byte raster according to the OSC specification. In PD patchers this is formatted automatically.

/TX +argument: = MIDI channel message (OSC type tag "m") alternatively:

/midi +argument: = MIDI channel message (OSC type tag "m")

Transmits an arbitrary MIDI channel message
According to the OSC specification the data contains of a group of 4 bytes:
The 1st byte selects the MIDI port – is ignored by this USB / WiFi / MIDI interface
The 2nd byte contains the MIDI status byte inclusive MIDI channel
The 3rd byte contains the 1st MIDI data byte (tone pitch, controller no., programm no., ...)
The 4th byte (if applicable) contains the 2nd MIDI data byte (velocity, controller value, ...).
If the status byte is received as 4th byte (as sent by "TouchOsc Brigde"), this is adjusted automatically.

/SSn (no argument)

Triggers tansmission of user edited string no. n (details see below) from MIDI OUT and USB

Data received at MIDI IN

are automatically transmitted via network as a stream of OSC messages. Depending on configuration parameter M, messages are styled differently:

--- M= 0: all bytes received at MIDI IN are sent as a continous stream of OSC "blobs", no care for syntactic correspondence of any bytes.

--- M= 1, 2 or 3: **MIDI channel messages and System Common Messages** (except SysEx) are sent as OSC m tag formatted /TX or /midi packets. Received bytes in "running state" are completed with the corresponding status byte. **SysEx** messages (any starting with 0xF0 and ending with EOX=0xF7 and max 256 bytes long) are sent as coherent OSC b tag formatted "blobs". Everything which cannot be put into this scheme or when synchronisation gets lost is forwarded as a continous stream of OSC "blobs" until a new synchronisation point is found.

/TX +argument: = OSC "blob"

The input buffer of the MIDI IN interface is polled in short time intervals. All bytes received since last polling (max.255) are packed into the OSC-"blob". The payload count is declared in a preponed 32 bit integer as specified by OSC. The end of the data block is filled up with additional null bytes to fit into the 32 bit raster according to the OSC specification. This way, data received at MIDI IN are effectively forwarded as a stream.

A message containing a NOTE ON message would look in the OSC packet as follows, for example:

```
/TX<0>  ,b<0><0>  <0><0><0><3>  <144><55><64><0>
```

This data format is not limited to MIDI channel messages, any arbitrary Sysex messages, strings and even single bytes can be transferred.

/TX +argument: = OSC "m" tag

This is the standard format when parameter M is set to 1.

Every single MIDI channel message is sent as a short OSC message with a single m tag. As long as possible, incoming bytes are interpreted as MIDI Status Bytes to start a new message or interpreted as MIDI Data Bytes, as long as possible as in "running state. If a new status byte is received before the last message has gathered the corresponding number of data bytes, the incomplete message is cancelled and the new one initiated.

A message containing a NOTE ON message would look in the OSC /midi packet as follows, for example:

```
/TX<0>  ,m<0><0>  <0><144><55><64>  ( M = 1)
```

/midi +argument: = OSC "m" tag

This format variant is specially contributed to be used with "TouchOSC MIDI bridge. Details see page20. When this message format is selected, it is not possible to distinguish between MIDI IN and USB !

A message containing a NOTE ON message would look in the OSC /midi packet as follows, for example:

```
/midi<0><0><0>  ,m<0><0>  <0><144><55><64>  ( M = 2)
/midi<0><0><0>  ,m<0><0>  <0><64><55><144>  ( M = 3)
```

If the interface is connected with "TouchOSC MIDI Bridge", the configured setting is temporarily overridden and the behaviour of M=3 is executed. See page 20.

Data received via USB

are automatically transmitted via network as a stream of OSC messages. Depending on configuration parameter M, messages are styled differently:

--- M= 0: all bytes received via USB are sent as a continous stream of OSC "blobs", no care for syntactic correspondence of any bytes. The method name is TU to distinguish the packets from data received at MIDI/COM

--- M= 1, 2 or 3: MIDI channel messages and System Common Messages (except SysEx) are sent as OSC m tag formatted /midi packets. Received bytes in "running state" are completed with the corresponding status byte. **SysEx** messages (any starting with 0xF0 and ending with EOX=0xF7 and max 256 bytes long) are sent as coherent OSC b tag formatted

"blobs". Everything which cannot be put into this scheme or when synchronisation gets lost is forwarded as a continous stream of OSC "blobs" until a new synchronisation point is found.

/TU +argument: = OSC "blob"

The input buffer of the USB interface is polled in short time intervals. All bytes received since last polling (max.255) are packed into the OSC-"blob". The payload count is declared in a preponed 32 bit integer as specified by OSC. The end of the data block is filled up with additional null bytes to fit into the 32 bit raster according to the OSC specification. This way, data received via USB are effectively forwarded as a stream.

A message containing a NOTE ON message would look in the OSC packet as follows, for example:

```
/TU<0>  ,b<0><0>  <0><0><0><3>  <144><55><64><0>
```

This data format is not limited to MIDI channel messages, any arbitrary Sysex messages, strings and even single bytes can be transferred.

/TU +argument: = OSC "m" tag

This is the standard format when parameter M is set to 1.

Every single MIDI channel message is sent as a short OSC message with a single m tag. As long as possible, incoming bytes are interpreted as MIDI Status Bytes to start a new message or interpreted as MIDI Data Bytes, as long as possible as in "running state. If a new status byte is received before the last message has gathered the corresponding number of data bytes, the incomplete message is cancelled and the new one initiated.
A message containing a NOTE ON message would look in the OSC /midi packet as follows, for example:

```
/TU<0>  ,m<0><0>  <0><144><55><64>  ( M = 1)
```

/midi +argument: = OSC "m" tag

This format variant is specially contributed to be used with "TouchOSC MIDI bridge. Details see p.21.
When this message format is selected, it is not possible to distinguish between MIDI IN and USB !
A message containing a NOTE ON message would look in the OSC /midi packet as follows, for example:

```
/midi<0><0><0>  ,m<0><0>  <0><144><55><64>  ( M = 2)
/midi<0><0><0>  ,m<0><0>  <0><64><55><144>  ( M = 3)
```

If the interface is connected with "TouchOSC MIDI Bridge", the configured setting is temporarily overridden and the behaviour of M=3 is executed. See page 20.

Editor for user defined strings **(to be triggered by OSC command/method /SSn)**

active in configuration mode (when **only** Jumper J2 is placed, virtual COM port via USB).

This feature is useful to trigger transmission of System Exclusive or similar messages, which are used repeatedly by user input. User defined strings are stored permanently in flash memory, ie. will be present after power cycle or system reset. Any string may be deleted or overwritten up to 10000 times.

ES n <return> starts the editor for string entry no. n (n =0 to 127).

The "entry number" n is entered as a one to three digit decimal number in ASCII text format and **terminated with <return>**. Next the user defined string is entered.

A "string" usually contains "printable letters", which have a correspondence on a typewriter or similar keyboard. "**Nonprintable letters**" - represented by a binary byte - may be entered as follows: first enter a backslash \ next enter the binary value as a 2 digit hexadecimal number (without type specifier like 0x). Leading zeroes for byte values less than hex10 have to be entered explicitly!

Each string may contain max. 253 characters (nonprintables=3 characters).
String input is terminated by <return>. Consequently, if the string shall contain a Carriage Return or a Backslash, this has to be entered as a binary hex byte as described above.

ES n <return><return> deletes string no. n (n =0 to 127). Technically this is a redefinition of the string with a zero length content. "Editing" of existing strings is not possible.

CS n <return> returns string no. n (n =0 to 127) **for check via control interface**.
If no valid user string is stored, the sequence n:xxx is responded.

LS responds a list with all valid string entries. After each 20 lines, the output stops and has to be retriggered with a <return> keystroke

Compiler for user defined OSC-commands ("methods")

active in configuration mode (when **only** Jumper J2 is placed, virtual COM port via USB).

OSC has the disadvantage (in contrast to MIDI) that no general command set is specified. So any OSC application has its individual set of commands.
To compensate this disadvantage, the USB / WiFi / MIDI interface is supplied with a simple compiler to generate user specific OSC commands/methods. Up to 128 methods may be created this way, each of them may call a complete sequence of OSC commands as described above. The compiler is active as a specially operated feature in the OSC command mode. Input is case insensitive, letters are always converted into upper case (except content of inline characters and strings to be transmitted at runtime).
At runtime, user specific methods/commands may be mixed arbitrarily with predefined commands described above. **User defined commands are generally evaluated with higher priority than those which are preprogrammed in firmware.** This provides an option to redirect a method/command which usually starts a predefined command into a user defined one with the same name.

Every received OSC command has to be compared with all user edited name entries. This takes some time of computation: at least a few microseconds, but in worst case up to 10 milliseconds. To keep search time low, put all user defined commands – particularly urgent ones -- into entry slots with low numbers. Short command names with quite different first letters will shorten the search time.

Path descriptions: Together with user defined command names path information like /../ is handled the same way as described above for built-in methods/commands.

EM n <return> starts the compiler for command entry no. n (n =0 to 127).

The "entry number" n is entered as a one to three digit decimal number in ASCII text format and **terminated with <return>**.
Next enter your command/method name (without preceeding slash, which is inserted automatically) including the "item number". If desired, the name may contain an OSC path styled like ../. Spaces will get part of the command name.
Instead of a definitive "item number" the star character * may be used as a wildcard. If the command name is terminated by the star *, during runtime parsing of every command name is terminated here. All following letters are ignored. This means especially, that the following user defined command sequence will be performed at runtime with every "item number", which may be copied into the user sequence by the same wildcard (see below).

The name input (including "item number") **is terminated by a comma**.

Next, an appropriately structured sequence of ASCII written commands is entered. Only those method/command names which are predefined in the firmware are allowed. **No recursive call** of user defined command sequences is possible. The command name is followed by obligatory and voluntary arguments as described above. In the compiler, arguments are entered as numerical ASCII text or as a placeholder (details see below). The first argument follows directly after the method/command name. Additional arguments are automatically separated by special characters. Any non-numerical ASCII character terminates the input of an argument. Examples see below. **The compiler entry is terminated by <return>**. Max. 253 characters per entry are allowed, incl command name.

EM n <return><return> deletes command entry no. n (n =0 to 127)

Due to the nature of flash memory, every command entry may be deleted and rewritten up to 10000 times. User commands may be overwritten with differing ones without beeing deleted explicitly before. "Editing" of existing methods/commands is not possible.

CM n <return> returns command sequence no. n (n =0 to 127) **for check via control interface**. If a nonconfigured command number is checked, the text "/xxx" is responded.

LM responds a list with all valid user defined methods. After each 20 lines, the output stops and has to be retriggered with a <return> keystroke

Parameters of command sequences are entered to the compiler as follows:

The leading slash / of every OSC method/command name is inserted automatically.
Enter method/command name to be executed next:

Most simple case: if no additional "opcodes" follow the redirected method, **all originally received arguments are adopted** from the originally received argument list to the redirected one. Compare with example at # below.

Example: EM1<return>/ROTARY5,/CC7<return>
At runtime it directs every OSC message from ROTARY5 method – explicitly the float argument – to CC7 method. This way the function of ROTARY5 is changed into a continous MIDI volume control.

More than one command name may be added in a user compiled method/command, these will be worked off in order of appearance including their attached arguments. But the user has to decide how far this makes sense for him.

Apart from inserting command names, **the compiler roughly works as follows**: when a received OSC method is "grabbed" at runtime by the interpreter for user compiled OSC methods, first all of its arguments are stored in a temporary argument buffer. Next for the user defined OSC method another temporary argument buffer is opened at runtime. "Temporary" means: valid until this user compiled method is worked off during runtime. **The set of "opcodes"** (as described below) **first transfers arguments** from the argument buffer of the received OSC method to the new generated in sequence of request. This means, in general the received OSC method must contain as many arguments (max. 5) as will be consumed by the corresponding user generated one. In practice this capacity is usually limited.

_ (underline) **prevents adoption of original arguments to a user defined method**

(only to prevent adoption completely. A problem may arise, when mandatory arguments are missing. For custom construction, preferably use operations described below)
Example: EM3<return>/PUSH1,/NO63_<return>
PUSH1 triggers transmission of NOTE ON 63 with default velocity. The argument of PUSH1 is ignored.

*** append received item number as item number to a user defined command**

Example: EM4<return>/ROTARY*,/FADER*<return>
At runtime it directs every OSC message from any ROTARY method – explicitly the float argument – to the corresponding FADER method with the same intem number.

= append accumulated Rotary Sum as item number to a user defined command

Example: EM4<return>/ROTARY3,/FADER=<return>
At runtime it directs every OSC message from ROTARY3 method – explicitly the float argument – to the corresponding FADER method with the accumulated Rotary Sum as intem number.

 # insert next originally received argument as next argument to actually sent message

Exclusively integer and float arguments can be "consumed" this way. Other argument types like blob, string and MIDI are ignored and the next appropriate argument of the originally received list is used.

Example: EM43<return>/ROTARY5,/CC7#&9 <return>
At runtime it directs every OSC message from ROTARY5 method – by use of "opcode" # explicitly the float argument – to CC7 method. The optional second argument &9 transmits MIDI channel no. 9.

The following "opcodes" are no placeholders to insert received arguments to OSC methods during runtime, but insert static values to arguments into user defined methods (optional arguments for example) or trigger immediate transmission of different kind of data from MIDI OUT and USB:

& insert inline decimal number as next argument

Inserted as mandatory or optional argument with the respective method name. This differs from opcode % as far as the argument may be "built" into the method.
See example above

@ insert inline hex number as next argument

Inserted as mandatory or optional argument with the respective method name. This differs from opcode % as far as the argument may be "built" into the method.
See example above

(insert accumulated Rotary Sum as next argument

This is hermaphrodite in the sense that an argument which is variable during runtime is inserted statically.

Example: EM7<return>/PUSH1,?/CC7(<return>

At runtime this example directs every OSC message from the PUSH1 method to the CC7 method. The PUSH1 button works as a trigger now to send a CONTROL CHANGE message to controller number 7 with value given by the actually accumulated settings of ROTARY1-5. The question mark before /CC7 causes transmission of the CC message only when the button is pushed down, not when it gets released. See "opcode" ? below.

~ (tilde) **replace (fake) the Rotary Sum by inline decimal number**

exclusively in a user compiled command string. This simplifies and expands capabilities of some commands/methods described here. Replacement is valid until another accumulating ROTARY command is received or **the last accumulated value is restored by the opposite "opcode" ^.**
Example: EM3<return>/BUTTON16,?~16/SC ^ <return>
/BUTTON16 sets the default MIDI channel to 16 and then restores the accumulated Rotary Sum.

% send decimal inline byte immediately

$ send hex inline byte immediately

Example: EM4<return>/PUSH1,?$F8 <return> triggers transmission of a single MIDI Timing Clock message

> send accumulated Rotary Sum as MIDI byte

" send inline string until next ". Enter nonprintable byte as \ followed by 2 digit hex number

Example: EM2<return>/PUSH1,?"\F0\7Dhello world\F7" <return>

:n (colon) **send user edited string no. n as MIDI sequence**

this replaces (i.e.simplifies) use of method /SSn in user edited command sequences.

; (semicolon) **send user edited string no. = (accumulated Rotary Sum) as MIDI sequence**

The following "opcodes" provide some limited options to handle flow control:

- (minus sign) **decrease argument pointer**

"Opcodes" #, (,< and ? each copy one argument from the originally received argument list and append it to a temporary argument list for the actually sent message (method). Then the argument pointer of the originally received argument list is increased to point to the next originally received argument.
By means of the minus sign "opcode" – the argument pointer is decreased by one. This may be useful in some cases to use an originally received argument for several messages.
Example: EM1<return>/FADER1,? **–** /NO65# **!** NZ65 <return>
transmits NOTE ON with velocity sent with the FADER method, but NOTE OFF instead of NOTE ON with zero velocity.

+ increase argument pointer

this increases the argument pointer by one, possibly to leave out one originally received argument from retransmission. Only useful in very special cases.

? conditional branch of execution: if next argument

= TRUE i.e. unequal 0: execute following sequence until ! or end. Exit then
= FALSE i.e. = 0: jump after ! and execute following until end

! marks end of TRUE branch after ?

If ? is compiled as "opcode" into a command sequence, at runtime the next orignally received OSC integer- or float-argument will be checked **and "consumed"** (i.e. argument pointer will be increased):

--- if it is equal to 0, execution of the command sequence will be interrupted at this point, **the list of commands will be scanned until entry ! is found**. From here the sequence will be executed normally until end of the command sequence.
If no ! does follow, execution will be terminated immediately.

--- If it is unequal to 0 (i.e. unequal 0.000 !!), execution of the sequence will be continued normally with subsequent command entries until "opcode" ! or the end of the sequence is found. Execution of the list will be terminated at this point.

By use of this feature it is possible, that different actions specifically **take place during "push" or "release" of a "Push"-or "Toggle-" object**, or release of the button becomes ignored totally.

--- "opcode" **! may be put directly after ?**. Commands which are put after the sequence ?! will be performed only when the button is released.

All commands before the question mark are executed independently of the value of the OSC argument checked by ?**.**

Attention, unconsidered use of control characters ? and ! may cause hard explainable malfunction !
During runtime, the question mark "consumes" an OSC argument. This is easily overlooked, when the sequence has to evaluate several arguments at runtime.
When a button is released, placeholders of the type #, which are put between the ? and ! mark, are not evaluated. **If necessary, at runtine OSC arguments are "deleted" by the + "opcode" or used multiply by the – "opcode".**

For example and as a practical assistance, the following **4 methods are programmed together with the firmware,** but can be changed by user input:

EM1: /TOGGLE1,?/SB

EM2: /TOGGLE2,?/SC

EM3: /TOGGLE3,?/SV

EM4: /TOGGLE4,?/SP/SA (take care of the "Rotary Sum" depending effect of /SP !)

Use as a virtual MIDI/WiFi cable together with TouchOSC MIDI Bridge:

In summary, for general use with buttons and faders, the common OSC messages with f and i tags work more reliably and easily than the TouchOSC Bridge, but sometimes it is good to have a MIDI software compatible interface for the Ethernet.

A more reliable and comfortable wireless MIDI control may be established, when a second piece of the board described here is used as USB to WiFi interface. Without additional hardware, for slow applications with moderate data transfer and uncritical latency, the MIDI2UDP quasi driver may be used (see <www.midi-and-more.de/midi2ether.htm>). Another very good solution is the ipMidi driver (see <www.nerds.de>), but unfortunately the RN171 module does not support UDP multicast.

First you have to install the "TouchOSC Bridge" software package. When you start a MIDI software then, you will find a MIDI IN and MIDI OUT port named "TouchOSC Bridge". Select this.

Before starting TouchOSC Bridge.exe, some details have to be cleared at the WiFi / MIDI interface:

To get into contact with an external device, **TouchOSC MIDI Bridge initially uses UDP broadcast and sends an OSC-formatted /ping message** to port 12102. If this is responded by a complementary UDP OSC formatted /ping message to the local IP of the asking PC, port no. 12101, the connection gets established. From then on, communication goes 1:1 UDP. Other connection relevant messages like mDNS (used by Bonjour for example) are not needed for this procedure.

So necessarily the **UDP host IP** ("set i h < > ") of the WiFi/ MIDI interface has to be directed to the PC where TouchOSC Bridge will be active. With standard setup of the RN171 in AP mode, the remote IP is usually set by DHCP to 192.168.1.11. The **local port of the RN171** ("set i L < > ") must be set to 12102, the **remote port** ("set i r < > ") has to be set to 12101. Last not least an active network connection is necessary. When all these details are ready, start TouchOSC Bridge. The WiFi / MIDI interface then automatically responds the /ping message and establishes the connection. **The /ping message from TouchOSC bridge is sent only once!** If it is missed by the WiFi / MIDI interface, you may have to restart your PC.

TouchOSC Bridge sends and receives OSC m tag formatted MIDI packets by a specific OSC /midi method as described just below. When the connection is established once, it stays active even when you change your MIDI software. Even MIDI System Common Messages (with Status Byte 0xF.., except System Exclusive and EOX) are handled correctly by TouchOSC Bridge.

In contrast to the m tag format as generally described in the corresponding literature, **TouchOSC bridge sends and expects the status byte as last byte in the message**. For incoming /midi messages, the WiFi / MIDI interface detects the format automatically. **For /midi messages sent to TouchOSC Bridge, the appropriate format should be configured (parameter M= 3).**

This is set temporarily when the /ping or /midi method is received and subsequently set / cleared in correspondence with the byte order of every received /midi message. If any packet with /TX method is received, the format of subsequently sent MIDI IN related OSC packets is temporarily reset to the format configured by parameter M --- and re-reset to TouchOSC Bridge format by every packet with the /midi method … and so on.

Apart from all of this, a TouchOSC app installed on the IPad distinguishes automatically between the OSC connection and MIDI Bridge connection. So a control on the IPad designed for MIDI by the TouchOSC editor automatically sends common OSC messages over the path which is specified for OSC too and sends both a common OSC and a /midi message over the same network connection when this is configured for both methods.

Appendix

Structure of MIDI channel messages (super compact crash course)

The first byte is the **status byte, only this one has bit 7 set**.

It is composed of 2 hex nibbles. The most significant (high) nibble describes the message type, the least significant (low) nibble contains the MIDI channel. The MIDI channel is always physically coded one less than written in any MIDI manual or litterature. Putting both together: (MIDI channel-1) has to be added to the message type status body.

For MIDI channel no. 1 the different types of MIDI messages show following status byte (=status body):

NOTE OFF:hex80,dec128
NOTE ON:hex90,dec144
POLY KEY PRESSURE(=POLY KEY AFTERTOUCH):hexA0,dec160
CONTROL CHANGE:hexB0,dec176
PROGRAM CHANGE:hexC0,dec192
CHANNEL PRESSURE:hexD0,dec208
PITCH CHANGE:hexE0,dec224

The status byte of PROGRAM CHANGE and CHANNEL PRESSURE is followed by exactly 1 data byte
The status byte of all other message types is followed by exactly 2 data bytes

In all data bytes the most significant **bit7 is cleared**, the other bits are content specific.

Determination of MIDI note values from common note names:

	C	C#	D	D#	E	F	F#	G	G#	A	B	H
-2	0	1	2	3	4	5	6	7	8	9	10	11
-1	12	13	14	15	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31	32	33	34	35
1	36	37	38	39	40	41	42	43	44	45	46	47
2	48	49	50	51	52	53	54	55	56	57	58	59
3	60	61	62	63	64	65	66	67	68	69	70	71
4	72	73	74	75	76	77	78	79	80	81	82	83
5	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107
7	108	109	110	111	112	113	114	115	116	117	118	119
8	120	121	122	123	124	125	126	127	-	-	-	-

This scheme is used by several MIDI sequencers. But deviating names of the different octaves are in use, too. Following unique relationship may be helpful: the concert pitch a (440Hz) -- in the table above "A3" - is always corresponding with MIDI note value 69 (hex 45).

Wireless DMX

is made possible with two USB to WiFi+MIDI/RS232 Interfaces.

The DMX data from a PC based control software should be fed to the wireless "transmitter" via USB (rotary switch pos.3). This **RN171 "transmitter" module** has to be configured as "access point" then, DHCP=SERVER, UDP protocol. The rotary switch of the wireless **RN171 "receiver"** has to be put to pos.F and this module has to be configured in infrastructure mode, associated with the other RN171 module, DHCP=ON, UDP protocol. The baudrate between each interface and its RN171 module should be 230400.

Wireless transmission with **XBee modules** makes configuration and operation much easier. The baudrate between each interface and XBee module should be 115200. The rotary switch of the PC sided interface has to be in pos.3, the DMX sided one in pos.7. This is sufficient for transfer of 256 byte DMX data packets with 30ms repetition rate (DMX Control 3.0 standard).

contact: wschemmert@t-online.de