### Design

#### Communication from Controller to slaves

##### Comparison

Many options are available for the communication between the controller and slaves. The most important is REQ Gen1.

Table : Communication Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| **Item \ Type** | **Cables** | **WIFI** | **RF** |
| Cost | - | ++ | +++ |
| Setup Time | --- | +++ | +++ |
| Speed | +++ | + | ?? |
| Reliability | ++ | - | + |
| Software | + | -- | -- |
| Extra connections | --- | +++ | +++ |
| Soldering work | --- | ++ | ++ |

Cables have as highest advantage, that these are reliable. However, all devices need to communicate with the Controller, so it will mean one cable per slave. And either two connections per slave to make some kind of network of many connections on the Controller. When multiple slaves of the same type should be connected, a network is the only possibility.

TODO: additional checks are needed:

* For cables:
  + What kind of protocol: SPI (already used by SRAM/SD) or I2C?
  + What connectors (Dsub?)
* For WIFI:
  + Needs for a router or can a local network be created?
  + Is speed high enough?
  + ESP8266 need to be programmed probably
* RF:
  + Check in live situations with lots of noise/other RF sources
  + Is speed high enough?
  + Free channel checking need to be implemented

##### Network Topology

The following requirements are needed for the network:

* The Controller sends messages to each slave device.
* Each slave device sends messages to the Controller.
* Slave devices do not send messages to other slave devices.

##### Slave Presence

Initially, the Controller needs to know which slave devices are present. Therefore, each device has its own predefined channel. TODO: how to define this / (auto) changing of channels in case of wireless communication.

During the initialization the following actions take place:

* The Controller polls for used devices/channels
* Each device returns its type (e.g. MIDI, Pedals/Switches etc)
* The Controller sends information to each device for what messages to send data back. This depends per slave.

##### Loop

After initialization, again the Controller always takes initiative. The reason is that important messages (like from MIDI) should be handled before lower priority messages (like a message from a pedal/switch) and important messages should not be interfered by possibly colliding messages from less priority slave devices.

Therefore the Controller polls each device for messages to be received. Maybe high priority devices will be polled more often than less priority devices.

Assuming all devices have same priority, the following flow will be typical for two slave devices:

1. The controller asks slave 1 if there are messages (Controller is transmitter)
2. The slave sends messages back (including a last (empty) message) (Slave 1 is transmitter)
3. Assuming these are high priority messages, messages are sent back. It can happen that some messages are discarded (e.g. MIDI CC messages with the same CC value).  
   It should be prevented that switching the transmitter happens too often).
4. Next slave is being handled (note that high priority slaves might be polled more often).

##### RF Proof of Concept

There are different types of RF, like below 1 GHz and above 1 GHz frequencies. Normally, 2.4 GHz is used. The default module to be used is NRF24L01+, however also the cheaper SE8R01 is available. Both will be used for the proof of concept.

* Which RF
* Used communication method: SPI
* Speed: Highest reliable / highest possible (cost more energy, but since an adapter will be used, this is not a problem). Default speeds are 250 mbps, 1 gbps, 2 gbps.
* Switching between receiving and transmitting.

##### NRF24L01 / BE8… TODO

These are the typical device used for 2.4 GHz wireless communication. Since the latter is cheaper and compatible, the latter is chosen. Also, it has a special network topology, using a 6-1 network for 6 transmitters and 1 receiver. The idea is to use this to:

* Let each slave be a transmitter
* The controller is the (only) receiver
* Each slave sends messages as soon as an input signal is received that has to be sent to the controller.
* After each message sent by a slave, the slave

##### One or Two RFs per device

After a short check it turned out that sending a message can take from 0,6 ms (16 byte payload) to several ms when retries are needed. Without retries sometimes messages are lost. However, several ms is too much to delay a message. Therefor an own protocol will be used.

Also to prevent switching (which takes 250 to 350 us excluding overhead) it is best to use 2 RF’s per device, one for sending. This means that devices can both transmit and receive without switching.

Also this makes it easier to send whenever needed, in case of collisions do a (smart) resend.

##### Best Channel / Initialization

The 2.4 GHz band used by the nRF24L01+ radios is from 2.4 GHz and has 125 possible channels. From these channels until channel with frequency 2.484 GHz is used by WIFI and therefore unusable by Mestra (also because WIFI will be heavily used by changing audience and thus very unpredictable).

The controller will check which frequency is best and send it to all slaves which can send a message back when received to move to the new frequency.

TODO: Check if the used frequency needs to be changed during operation.

This way the controller automatically knows which devices are present (the slave devices return their type).

##### Normal Operation

Devices will send messages when available. This means it can happen that messages collide. If so, there is a default scheme how long each device waits until trying to retransmit (varying from 10 us to 100 us).

The time to send a 8 byte payload message cost (see datasheet, page 38, fragment below).

Since ACKs will not be used, the time will be T\_UL + 2 \* T\_stdby2a + T\_IRQ = PL / SPI\_data\_rate + 2 \* 0,130 us + 0,006 us =113 / 16 us + 2 \* 0,136 us = around 7 us. However when tested, the shortest time was around 136 us (so let’s assume that).

Where PL (payload length) = 8 \*( 1 + 3 (address) + 8 + 1 (CRC) )+ 9 (bits) = 113 bits.

The SPI rate is assumed 16Mbps

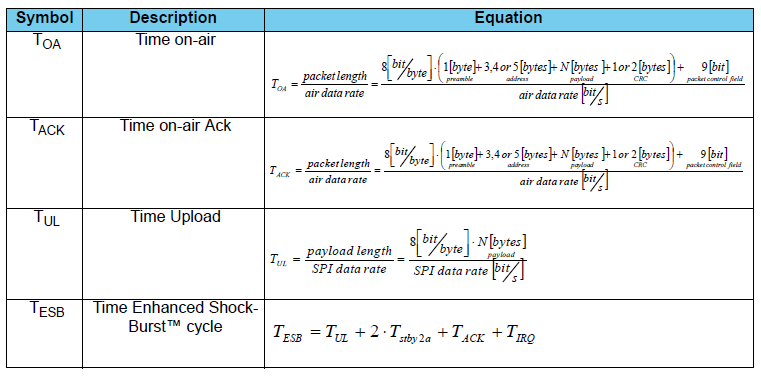


Figure : Time Equations

* This means per second 1000000 / 136 / 2 = 3676 messages can be sent. The division by 2 is because for each message an acknowledgement message is sent. Retries are not taken into account here.

##### Acknowledgements

Every message has to be acknowledged, both from controller and from slaves. An acknowledgement command has two bytes: one Ack ID and the sequence number to be acknowledged. The other 6 bytes can be used for normal messages which otherwise had to be sent normally.

Acknowledgements need to be sent less than 0.5 ms after receiving the message, to prevent the sender sending an equal message, thinking it has not arrived.

##### Reducing Messages

To reduce messages multiple commands can be stored in one payload.

Also messages to be sent are not sent within a 2 ms from one device. Messages are ‘saved’ until ready to be sent after 2 ms.

##### Multiple Messages

All commands either have a fixed length or their length inside the command as first byte(s) after the command ID. This means the receiving device knows if a message is unfinished and has to wait for more. In this case an ACK is sent only after the last one (last message). This prevents collisions and reduces messages.

##### Retries

It can happen that retries are need. For example, when the receiving device does not get a message, or when an acknowledgement message is not received back. 2 ms after sending a message and not getting an acknowledgement message, the message is sent again. An acknowledgement message does not need to be acknowledged, since the sender of the original message will send its message again after 2 ms, and continues to do so until the message acknowledge has been received.

Note that it takes about 0.5 ms to send a message, the acknowledge message will be sent by the receiver after a very short time (< 0.5 ms) and sending takes another 0.5 ms. So 2 ms should be adequate to be sure the message has not been received.

##### Payload

This depends on the type of device. All messages are 8 bytes. In case bigger messages need to be sent, they are split up (e.g. MIDI system exclusive messages). The partial messages need to be stored until complete.

Device IDs do not need to be sent separately, but a message ID is needed for the acknowledgement. Therefore each first byte of the payload is the sequence number for that device message counter.

The other bytes are e.g. for a MIDI device, Note on command: 2nd byte: NoteOn, 3th byte: Note number, 4th byte: velocity.

Multiple MIDI messages can be packed in one payload.

##### Alive Message

If the controller has not been sending a message to a slave for 5 seconds, it will send an alive message. If there is no reaction, both the controller and slave will notify it with a LED.

#### Diagnostics LEDs

All devices will have a LED to show the power. The reason is that almost all boxes will not have any direct notification if it has power or not.

Also, each box communicates through RF, so for this reason a LED will show the RF status. Note that when the controller polls each device, this will not be shown (since it happens every few ms), and slave devices receiving a poll request for packages and no messages are available (thus an ‘empty’ ACK package is sent, will also not result in a LED being on at the slave side, since this will happen many times per second.

Table : Generic Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with other device(s).  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

Blinking details:

* Slow blinking means one blink of 20 ms, followed by a pause of 980 ms (totaling 1 s).
* Double fast blinking means two blinks of 20 ms with a 80 ms gap, followed by a pause of 800 ms (totaling 1 s).
* Triple fast blinking means three blinks of 20 ms with a 80 ms gap, followed by a pause of 700 ms (totaling 1 s).

## Generic Software

### Requirements

This paragraph shows all requirements common to all Mestra (embedded) software.

Table : Requirements GS

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Base ID** | **Version** | **Category** | **Item** | **Description** |
| GenS100 | - | 1.0 | Rules | Amount | At least 100 rules can be defined.  *Rationale: to have a useful system, enough programming flexibility is needed.* |
| GenS200 | - | 1.0 | Rules | Simultaneously | At least 20 rules can be active simultaneously.  *Rationale: to have a useful system, enough programming flexibility is needed.* |
| GenS210 | - | 1.0 | Triggers | Amount | At least 100 triggers can be defined.  *Rationale: to have a useful system, enough programming flexibility is needed.* |
| GenS220 | - | 1.0 | Commands | Amount | At least 100 commands can be defined.  *Rationale: to have a useful system, enough programming flexibility is needed.* |
| GenS230 | - | 1.0 | Commands | Per Trigger | At least 10 commands per trigger can be defined.  *Rationale: to have a useful system, enough programming flexibility is needed.* |
| GenS300 | - | 1.0 | Communication | No noise from other devices | Slaves should work also when other (non Mestra) devices using the same type of communication are present.  *Rationale: when using wireless communication, other RF or WIFI should not interfere with Mestra devices.* |
| GenS310 | - | 1.0 | Communication | No noise to other devices | Other devices should not be interfered more than needed by the use of Mestra devices.  *Rationale: except for the channels/ports/resources used by Mestra devices, all other non Mestra devices using the same communication should not be interfered with.* |
| GenS320 | G20 | 1.0 | Communication | Protocol | The communication protocol for all devices will be similar.  *Rationale: Adding devices need to communicate with the controller.* |
| GenS400 | G1 | 1.0 | Performance | Prevent Messages | Prevent sending messages from a slave to the Controller when not needed.  *Rationale: Sending (and the resulting received message(s)) cost a lot of time, also it pollutes the bandwidth.* |
| GenS500 |  | 1.0 | Testing | Unit testing | Testing of software can be performed automatically by using unit testing.  *Rationale: Manual testing is too time consuming and software will be complex.* |

### Design

#### Slave Loop

Each slave has the same loop:

* If slave has an incoming message (from the controller):
  + Read incoming messages
  + Convert to specific signal/message for device
  + Send signal/message to device specific output(s)
  + Show notification (LED)
* If device has a incoming (device specific) signal/message
  + Convert to generic message
  + Send to Controller
  + Show notification (LED)

#### Preventing unnecessary messages

For GenS400, unnecessary messages should not be sent. Unnecessary messages are messages, which will not be changed by the Controller, and it is known beforehand.

However, for this requirement, some intelligence need to be moved to the slaves:

* Each slave should know for which type of messages sending to the Controller is not needed.
* In this case, the device possibly has to ‘forward’/return the message to its output(s).

Also, as an additional refinement, the device can be forwarded/return its message (without any change), BUT still send it to the Controller (to be processed further). For e.g. MIDI Note On commands this will ensure a better latency (while still the Note On can result in additional messages).

The refinement results in:

* The Controller needs to send to each device a list of messages to handle itself, send to the controller, or both.
* Each device needs to store this list and act upon it.
* Note that the list can be changed realtime (through the Controller).

#### IDE

The default IDE to be used for Arduino is the Arduino IDE. However, this IDE has the problem that when using more than approximately 10 files, the file names do not fit in the upper status bar. Therefore, another alternative will be chosen.

There are many options, like Eclipse, Microsoft Visual Studio plugins, and plugins for editors. Since I am used to Visual Studio, I selected the VisualMicro extension for Visual Studio.

#### Folder structure

The following folder structure will be used:

* Mestra
  + Documents
    - Manual
    - Design document (this document), in the future split up by device.
    - Excel sheet for design document
    - (Future) Ideas
  + Code
    - Project file for Visual Studio
    - Production
      * Common
      * Controller
      * Midi
      * Debug
    - Test
      * Common
      * Controller
      * Midi

## Testing

### Unit Tests

Because unit testing is needed in an automated way, classes should not use Arduino.h (or Arduino.h has to be stubbed). All tests will be performed within Visual Studio in a separate Test Project, runnable without hardware.

### Integration Tests

All devices will have a set of (manual) integration tests. Mostly this will involve the Controller.

# Application (AP)

## Requirements

Table : Requirements AP

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Base ID** | **Version** | **Category** | **Item** | **Description** |
| App100 |  | 1.0 | User | Friendly | The application should be user friendly.  *Rationale: Users want an easy way to define rules and are not always technical experts within the MIDI domain.* |
| App110 |  | 1.0 | Platform | Independency | The application should be supported for various operating systems, at least Windows and Mac OS.  *Rationale: Many users into music use either Windows or Apple (Mac OS).* |

## Design

### Programming Language

For Since the user interface of the external application will be GUI-heavy, a high level (object oriented) language is desired.

For REQ App110, a programming language needs to be used that support multiple platform. C# / C++ with XAML/WinForms are not good candidates, since Macintosh is not supported.

Java is a candidate. Possible GUIs are:

* TODO

Python is a candidate. Possible GUIs are:

* TODO

Below advantages between languages/GUIs can be checked.

TODO

Table : Comparison Programming Languages

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Java / |  |  | Python / TkInter |
| Experience |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# Controller

## Generic

### Requirements

None.

## Hardware

### Requirements

Table : Requirements CH

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Base ID** | **Version** | **Category** | **Item** | **Description** |
| ConH1 | Gen1, GenS1,  GenS10, GenS20, | 1.0 | External Memory | Amount | The memory size should be high enough to store all rules (triggers/commands).  *Rationale: To meet REQ Gen1, data cannot be loaded from a slow component (SD card).* |
| ConH2 | Gen2 | 1.0 | SD | Amount | The time to copy all rules to external memory should take less than 5 seconds.  *Rationale: another 5 seconds is left for software initialization and device (wireless?) coupling.* |
| ConH10 | Gen10 | 1.0 | Diagnostics | Error | Whenever an error occurs, LEDs will be used to show the root cause. |

### Inputs/Outputs

The controller will have the following inputs:

* USB, only used to flash the sketch.
* Adapter, for power.
* SD card, for the SD card containing the configuration file.
* Wifi (wireless), for wireless transmission of the sketch.
* RF (wireless), for receiving messages from slave devices.

The controller will have the following outputs:

* RF (wireless), for sending messages to slave devices.
* LEDs, for receiving/transmission and diagnostics information.

### Design

#### Arduino Type

Due to REQ ConH20, 2 KB for an Arduino Uno is on the low side, therefore an Arduino Mega is chosen.

Also, during development, it is comfortable to have the RX available for debugging, and having multiple UARTs is convenient.

Most important, the Arduino Mega has 256 KB flash instead of 16 KB. It is expected, the implementation of all commands, triggers and rules may be consuming way more than 16 KB.

#### External memory selection

Excel sheet <TODO> shows 128 KB is enough to store everything according REQ ConH1.

An Arduino Uno has 2 KB, an Arduino Mega 4 KB internal memory. Therefore, external memory is needed.

Two types of memory are taken into account:

Table : Comparison External Memory

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Item** | **3K256 SRAM** | **23LC1024 SRAM** | **LP16040 (TODO)** | **EEPROM** |
| Reading speed | +++ | +++ | - | +++ |
| Writing speed | +++ | +++ | + | - |
| Wear-off | +++ | +++ | +++ | --- |
| Ease of use | -- | +++ | --- | + |

The reading speed of LP16040 is low, since it is a DIP24 IC, which needs either lots of digital pins, or a shift register, resulting in low read/write times.

EEPROM has problems when needing to write often that is wears off the IC.

3K256 SRAM has only 32 KB and 4 ICs are needed, with logic to combine them.

Therefore, it is clear a 23LC1024 is the best solution.

#### SD Card

Due to CH210, the SD card should be copied to external memory within seconds.

The typical SD card speed using SPI on an Arduino is like 100 KB/s, reading 128 KB results in approximately 1 second, which is well within spec.

#### Communication between external application and PC

The configuration file generated from the external application needs to be stored on the controller. This can be done in multiple ways:

Table : Comparison Communication Ext Application/PC

|  |  |  |  |
| --- | --- | --- | --- |
| **Item** | **SD** | **USB cable** | **Wireless** |
| Ease of use | -- | + | +++ |
| Extra hardware needed | +++ | +++ | --- |
| Enclosure complexity | -- | +++ | + |
| Reliability | +++ | +++ | + |
| Usability | +++ | +++ | -- |
| Software complexity | +++ | --- | --- |

It shows a USB cable is the best option. However, making a SD opening in the enclosure of the controller, makes it possible to use both the SD and USB cable option.

The wireless option will be not implemented, since it might well be possible a wireless network is present (although very likely during creating the configuration file). But except the transferring of the configuration file, it has no further use. Also using an RX/TX signal, it might be problematic when using RX/TX for another wireless solution to/from the slaves (see next paragraph).

#### Diagnostics LEDs

Table : Controller Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF, one per slave (max 5) | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

#### Diagnostics/Errors

Available LEDs:

* On/Off (blue)
* Receiving (green)
* Sending (red)

Whenever an error occurs, LEDs will display as shown below.

Table : Diagnostics Controller

|  |  |  |
| --- | --- | --- |
| **Root Cause** | **LEDs** | **Solutions** |
| SD Card cannot be read | On/Off: Blinking  Receiving: Off  Sending: Off | Reinsert the SD card and repower the device. |
| SRAM not accessible | On/Off: Blinking  Receiving: Off  Sending: On | None. |
| RF not accessible | On/Off: Blinking  Receiving: On  Sending: Off | None. |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

The breadboard will contain the RF communication breakout board, the adapter for it, and the SRAM chip including LEDs. This will take up maximum a half bread board.

### Proto Layout

TODO

### Component List

The cost of a prototype will be approximately (see Excel document, tab Cost):

Table : Components Controller

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Item** | **Item Cost** | **Amount** | **Total** | **Datasheet** |
| Arduino Mega | € 7,35 | 1 | € 7,35 |  |
| SD Recorder Shield | € 1,50 | 1 | € 1,50 |  |
| SD Card 128 MB | € 3,00 | 1 | € 3,00 |  |
| Enclosure | € 5,00 | 1 | € 5,00 |  |
| SRAM | € 3,50 | 1 | € 3,50 |  |
| 2 x RF Transceiver | € 0,80 | 2 | € 1,60 |  |
| Various electronics | € 2,00 | 1 | € 2,00 |  |
| Enclosure | € 5,00 | 1 | € 5,00 |  |
| **Total** |  |  | **€ 28,95** |  |

## Software

### Requirements

Table : Requirements CS

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Base ID** | **Version** | **Category** | **Item** | **Description** |
| ConS1 | GenH10 | 1.0 | Flash | Amount | The amount of Flash memory is limited.  *Rationale: Cost perspective.* |
| ConS10 |  | 1.0 | SRAM | Usage | The total combination of triggers, rules and commands are only limited by the storage (there should be no predefined mapping).  *Rationale: More flexibility* |

### Design

The controller has by far the most extensive sketch. The reason for this is, that all intelligence is built into this device. It receives messages, transforms them, and sends them to the correct slave(s).

### Memory Usage

#### Introduction

A lot of information need to be stored. Because of REQ ConS10, the default 8 KB SRAM of the Arduino Mega is not enough.

The internal 8 KB SRAM will be used for only the highest necessary data (REQ ConS1)

* Stack trace (can be larger than other modules, due to the complexity of rules and the OO kind of programming)
* Buffers (initially 512 bytes for the SD card -> SRAM buffer, these buffers will be removed after copying).
* Local variables

Therefore, none other lists should be saved in SRAM. All lists will be stored in external SRAM (unless REQ Gen1 is affected).

The external 128 KB SRAM will be used for storing the rules and commands for all (slave) devices. To be most flexible, the SRAM will not be divided evenly per slave, but used as needed.

Also, memory consumption should be kept low.

In the following paragraphs, per slave device the memory layout/usage is explained.

#### Trigger filtering

The Controller will send messages to the slaves for which signals messages will be sent towards the Controller. This way the Controller will only get messages that really needs to be processed, reducing the number of packages to be sent by the RF network and keeping the processor of both Controller and slaves lower.

What needs to be stored are all trigger tables themselves, and all commands and all current states (because commands rely on them). Below per slave the memory usage/mapping will be explained.

#### Generic

TODO

#### MIDI

#### Pedals/switches

Assuming there will be 8 pedals and 8 switches, the size needed is 8 (switches) \* 2 (type on/off) \* 2 (address size) + 8 (pedals) \* 3 (types) \* 2 (address size) = 32 + 48 = 80 bytes, which is negligible.

### Timing Performance

TODO

## Testing

### Unit Tests

The Controller has the most complex software thus testing is critical.

The classes regarding rules, commands, triggers etc. are not using Arduino specific code. This means the tests can be created by running them on a standard PC.

### Integration Tests

The following tests will be performed (by connecting a keyboard/synthesizer):

1. Play a note, same note should be played.
2. Play many notes, all notes should be played, no notes should be heard after releasing all notes.
3. As 1, while moving the joystick.
4. As 2, while moving the joystick
5. As 1, while using aftertouch
6. As 2, while using aftertouch
7. Specific tests for rules, commands etc. TODO

RF Interference tests:

1. Testing with one or more mobile phones communicating via WIFI
2. Testing with one or more mobile phones communicating via Bluetooth
3. Testing while one or more mobile phones are starting to communicate via WIFI
4. Testing while one or more mobile phones are starting to communicate via Bluetooth
5. Putting a microwave near the controller or slave devices

# Audio

## Generic

### Requirements

TODO

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Diagnostics LEDs

Table : Audio Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

TODO

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

TODO

### Design

TODO

### Memory Usage

TODO

### Timing Performance

TODO

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# DMX

## Generic

### Requirements

TODO

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Diagnostics LEDs

Table : DMX Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

TODO

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

TODO

### Design

TODO

### Memory Usage

TODO

### Timing Performance

TODO

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# Drum Pad

## Generic

### Requirements

TODO

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Diagnostics LEDs

Table : Drum Pads Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

TODO

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

TODO

### Design

TODO

### Memory Usage

TODO

### Timing Performance

TODO

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# Drums Trigger

## Generic

### Requirements

TODO

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Diagnostics LEDs

Table : Drums Trigger Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

TODO

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

TODO

### Design

TODO

### Memory Usage

TODO

### Timing Performance

TODO

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# GUI

## Generic

### Requirements

TODO

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Diagnostics LEDs

Table : GUI Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

TODO

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

TODO

### Design

TODO

### Memory Usage

TODO

### Timing Performance

TODO

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# Microphones

## Generic

### Requirements

TODO

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Diagnostics LEDs

Table : Microphones Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

TODO

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

TODO

### Design

TODO

### Memory Usage

This device does not receive messages, since there are no outputs.

Microphone data will not result in much data. There will be a number of microphones (based on frequency / frequency range) and regions of volume or edge detection. Assuming there are 3 microphones, each microphone can send either a value change or edge detection up or down change command (3 types), resulting in 3 (microphones) \* 3 (types) \* 2 (address size) = 18 bytes (negligible).

### Timing Performance

Only the translation from signals to messages is needed, and sending them to the Controller:

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# MIDI

## Requirements

Table : Requirements MG

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Base ID** | **Version** | **Category** | **Item** | **Description** |
| MidG1 | GenH10 | 1.0 | Flash | Amount | The amount of Flash memory is limited.  *Rationale: Cost perspective.* |

## Hardware

### Requirements

Table : Requirements MH

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Base ID** | **Version** | **Category** | **Item** | **Description** |
| MidH100 | - | 1.0 | Connectors | Inputs | Multiple MIDI In connectors should be supported.  *Rationale: at least one for a main keyboard, secondary keyboard and foot controller.* |
| MidH110 | - | 1.0 | Connectors | Outputs | Multiple MIDI Out connectors should be supported.  *Rationale: Possibility to send MDI messages to multiple keyboards without the need to use another THRU connector.* |
| MidH120 | - | 1.0 | Connectors | Thrus | Each MIDI In connector should have its MIDI Thru connector. |

### Inputs/Outputs

TODO

### Design

#### Safety

Because of REQ GenH30, connected devices should not damage the device. Therefore, opto isolators will be used for the MIDI In connectors.

#### Opto Isolators Types

The following are options which have been checked:

1. 6N137: Default opto couplers, however, these are not fast enough for MIDI messages
2. 6N138: Faster, however, not selected since H1L11 opto isolators are adviced (see below); advised by MIDI protocol.
3. H11L1: Faster, see Arduino-stack exchange links <https://arduino.stackexchange.com/questions/39908/h11l1-opto-isolator-does-not-pass-signal-to-arduino-correctly> and <https://electronics.stackexchange.com/questions/311889/i-cannot-get-opto-coupler-h11l1-to-work-as-midi-input> ).
4. PC900: advised by MIDI protocol; not checked, since I already have 6N138s and H11L1s.
5. Multi channel opto isolators: since multiple opto isolators are needed (one for each MIDI channels, 3 in total), a three channel opto isolator would be ideal. However, there are dual opto isolators, but these are too slow for MIDI messages.

The selection is made for H11L1, although not as one of the advised opto isolators. These are used in MIDI circuits however, and work flawlessly. Also since these ICs are only DIP6 instead of DIP8 and need fewer components around (resistors/5V lines), H11L1 is the best solution. Also the resistors around the H11L1 can all be 220 ohm like in the advised MIDI circuit.

#### EMI/EMC performance

Since in a life situation EMI can result in problems, for all MIDI inputs/outputs inductor beads will be used. The advised are not to be found for a cheap price, however a test will be made with 3T 6\*10 R6H (six hole) magnetic beads which have similar stats (TODO) as the advised inductors from the MIDI protocol.

#### Amount of MIDI Connectors

Every MIDI In uses a Serial (RX). Tests using SoftwareSerial results in unreliable MIDI messages. Since this can result Note Off commands being corrupted/not received, this is fully unacceptable in a life situation. This happens only when lots of messages are received (e.g. during lots of CC messages like the modulation wheel/after touch). Therefore, SoftwareSerial is unusable for MIDI.

That means each MIDI In needs a hardware RX. Since the Arduino Mega is the only Arduino (working on 5V), giving 4 UARTs, an Arduino Mega is necessary. One UART possibly has to be used for the RF communication, leaving 3 MIDI Ins available.

Therefore, it is logical to have 3 MIDI Thrus and 3 MIDI Outs.

This satisfies REQ MidH110, MidH120 and MidH130.

#### Diodes Type

For the MIDI Ins, diodes are used. The most default used are 1N4001 diodes, which work (tested), however 1N4148 seem to be better suitable for fast performance, so 1N4148 diodes will be used.

#### Diagnostics LEDs

Table : MIDI Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |
| MIDI In | Green | Off: Not receiving MIDI data  On: Receiving MIDI data |
| MIDI Out | Green/Red (bicolor) | Off: Not transmitting MIDI data  Green: Transmitting unprocessed MIDI data  Red: Transmitting processed MIDI data |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

For the MIDI device, two breadboards are needed, since there fit only about 6 MIDI connectors on one breadboard. Since 9 MIDI connectors will be placed, the breadboards will contain the following items:

1. Breadboard 1 (830 holes breadboard)
   1. MIDI IN 1, MIDI OUT 1, MIDI THRU 1
   2. MIDI IN 2, MIDI OUT 2, MIDI THRU 2
2. Breadboard 2 (400 holes breadboard or half 830 holes breadboard)
   1. MIDI IN 3, MIDI OUT 3, MIDI THRU 3

### Proto Layout

TODO

### Component List

* Enclosure
* Arduino (compatible) Mega
* SD Data Logger Shield (with SRam)
* Arduino Uno Protype Shield, with optocouplers and other logic.
* 5 MIDI DIN female connectors
* 4 dual green/red 3mm LEDs (for MIDI ins and MIDI out)
* 1 green 3mm LED (for MIDI thru)
* 12V adapter
* 6 x 220 Ohm resistors
* TODO resistors
* Bypass capacitors: TODO
* 2 x 4N189 diodes
* 3 x 6N137 Optocouplers

## Software

### Requirements

Table : Requirements MS

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Base ID** | **Version** | **Category** | **Item** | **Description** |
| MidS100 | TODO | 1.0 | Messages | Prevent unnecessary | A mechanism to immediately return/forward messages should be implemented, and to not send messages to the Controller.  *Rationale: To prevent unnecessary messages (costing time/bandwidth).* |
|  |  |  |  |  |  |

### Design

#### MS100: Preventing unwanted messages

To prevent unwanted messages, lists are needed for what messages:

* will be returned/handled immediately by this device
* will be send to the Controller.

For both, one bit is needed.

Assuming 16 MIDI channels, 128 notes, this results in 16 \* 128 \* 2 bits = 512 bytes

For CC, there are 16 MIDI channels, 128 CCs, resulting in the same number of bytes.

This totals 1024 bytes, which is 1 KB. Since the Arduino Mega has 4 KB, this suffices.

### Memory Usage

#### Trigger conditions

All trigger conditions need to be stored. MIDI trigger conditions can be per MIDI channel, note or CC, its velocity/release velocity/value and per type (change, into a region, out of a region).

There can be a maximum of 64 MIDI channels (4 MIDI In’s with each 16 MIDI channels).

There can be 128 note or CC values.

There also can be many trigger regions, even for the same note or CC.

This would result in an endless number of trigger conditions. Therefore, a more flexible way is used:

* Per MIDI channel a table is created, containing:
  + A list which is generic for all notes.
  + A list per note
  + A list per CC
  + A list for all program changes
* Each condition also has a type (value change, out of a region, inside a region). Therefore the current values should be stored to be compared against.
* Only USED midi channels, USED notes/CCs and USED program changes are present in the list
* Trigger conditions for note velocities/release velocities and CC values are not possible; these will be handled by filtering the property inside command.
* The application (APP) will make sure that the maximum amount of memory can be used, by assigning all lists dynamically.

#### Note On/Off

Assuming 4 MIDI channels are used, there will be a list generically for all notes. This list can have multiple entries, for the entire range, or one or more specific ranges.

*Example: A Note On command for note C4 with velocity 80 on MiDI channel 2 is received. First the generic list for MIDI channel 2 is checked. All conditions within the list are compared and when a trigger matches, a message is sent to the Controller. Than all conditions for the specified note is checked against all conditions.*

The memory needed will be for e.g. 4 MIDI Channels, 1 generic list (per MIDI Channel) and assuming 50 note (including region) will be:

| **Category** | **Item** | **Size** | **Unit** | **For Memory Counting** | **Remarks** |
| --- | --- | --- | --- | --- | --- |
| General | Used MIDI channels | 64 | MIDI channels |  | Address, 0 if not used. |
|  | Bytes for used MIDI channels | 128 | MIDI channels | 128 |  |
| Notes | Used MIDI channels | 4 | MIDI channels |  |  |
|  | Notes per MIDI channel | 129 | Notes |  | Address, 128 notes + 1 generic |
| CCs | CC per MIDI channel | 129 | CCs |  | As above |
| Lists | Total tables | 1024 | Tables |  |  |
| Table | Assumed length | 30 | Items |  | 30 items |
|  | Item size | 3 | Bytes |  | Assuming 3 types (1 byte), start region value (1 byte), end region value (1 byte) |
|  | Table size | 90 | Bytes |  |  |
|  | Tables size | 92,160 | Bytes | 92,160 |  |

### Timing Performance

MS100 will reduce the number of messages.

Worst case MIDI latency is defined by the sum of:

1. Time to receive a MIDI message
2. Time to process it by the device
3. Time to send it to the Controller
4. Time to process it by the Controller
5. Time to receive a message by the Controller
6. Time to send it through MIDI.

For a typical message of 3 bytes, this results in:

1. MIDI has a baudrate of 31,250 baud, which results approximately in a receive/time of 0,768 ms
2. TODO
3. This will be calculated in TODO.
4. See 3
5. See 2
6. See 1

This totals to 0,768 ms + TODO + 0,768 ms = TODO.

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# Proximity

## Generic

### Requirements

TODO

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Proximity LEDs

Table : Audio Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

TODO

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

TODO

### Design

TODO

### Memory Usage

This device does not receive messages, since there are no outputs.

Only a buffer is needed for sending received signals from switches and inputs, which is TODO bytes.

### Timing Performance

Only the translation from signals to messages is needed, and sending them to the Controller:

TODO

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# Pedals/Switches

## Generic

### Requirements

TODO

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Diagnostics LEDs

Table : Pedals/switches Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

TODO

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

TODO

### Design

TODO

### Memory Usage

This device does not receive messages, since there are no outputs.

Only a buffer is needed for sending received signals from switches and inputs, which is TODO bytes.

### Timing Performance

Only the translation from signals to messages is needed, and sending them to the Controller:

TODO

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# Remote

## Generic

### Requirements

TODO

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Diagnostics LEDs

Table : Remote Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

TODO

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

TODO

### Design

TODO

### Memory Usage

This device does not receive messages, since there are no outputs.

Only a buffer is needed for sending received signals from pressed/changed controls, which is TODO bytes.

### Timing Performance

Only the translation from signals to messages is needed, and sending them to the Controller:

TODO

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# USB MIDI

## Generic

### Requirements

TODO

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Diagnostics LEDs

Table :USB MIDI Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

TODO

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

TODO

### Design

TODO

### Memory Usage

TODO

### Timing Performance

TODO

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

# Debug

## Generic

### Requirements

Table : : Requirements EG

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Base ID** | **Version** | **Category** | **Item** | **Description** |
| DbgG1 |  | 1.0 | Debug | Devices | Debugging from all devices will be possible, without using the normal communication (like RF).  *Rationale: using the normal communication might interfere the normal messages sent over it.* |
| DbgG2 |  | 1.0 | Debug | Amount | All devices can send debug messages to the Debug device.  *Rationale: Debugging should be possible over multiple devices.* |
| DbgG3 |  | 1.0 | Debug | Amount | The hardware RX pin should only be used for sending debug messages to the computer showing diagnostics messages.  *Rationale: Using the hardware RX pin for other purposes, would corrupt the messages.* |

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Software Serial pin layout

Since only one directional communication is needed, the following pin layout will be used:

* Pin 0 (RX): For hardware debug port/serial
* Pin 1 (TX): For hardware debug port/serial
* Pin 2 .. 9: 8 software serial RX pins for 8 devices
* Pin 10..13 can be used for LEDs (TODO)

#### Diagnostics LEDs

Table : Debug Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non-empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

The breadboard contains the LEDs as described in paragraph **Error! Reference source not found.**. This will take up a 400 holes breadboard or half 830 breadboard.

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

### Design

#### Arduino Type

Since only one hardware RX is needed, and the functionality is rather simple, an Arduino Uno will provide enough.

#### Software serials

To meet REQ DbgG3, for each device a Software Serial will be used. 8 can be used to debug 8 different devices. A loop will be made to iterate over all 8 software serial RX’s, and print it to the hardware serial.

### Memory Usage

Each software serial buffer uses 64 bytes, the hardware serial too, this means (8 + 1) \* 64 = 576 bytes for buffers.

### Timing Performance

When a lot of messages will be received, buffers can be full, however the debug messages are not critical.

## Testing

### Unit Tests

TODO

### Integration Tests

TODO

Appendix A: New device template

# New Device Template

## Generic

### Requirements

TODO

## Hardware

### Requirements

TODO

### Inputs/Outputs

TODO

### Design

#### Diagnostics LEDs

Table : NEW\_DEVICE Diagnostics LEDs

|  |  |  |
| --- | --- | --- |
| **Function** | **LED Color** | **Description** |
| Power | Blue (generic) | Off: Power off  On: Power on |
| RF | Yellow (generic) | Off: empty message transmitting/receiving  Slow blinking: contact with controller  Double fast blinking per second: no contact with slave  Triple fast blinking per second: problem with RF  On: non empty message transmitting/receiving |

Note that if the GUI Device shows errors whenever possible.

### Breadboard Layout

TODO

### Proto Layout

TODO

### Component List

TODO

## Software

### Requirements

TODO

### Design

TODO

### Memory Usage

TODO

### Timing Performance

TODO

## Testing

### Unit Tests

TODO

### Integration Tests

TODO