THRII System Exclusive Protocol

Disclaimer

DISCLAIMER: THE INFORMATION HERE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHOR OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE PROVIDED INFORMATION OR THE USE OR OTHER DEALINGS IN THIS INFORMATION.

I have absolutely no connection to Yamaha / Line6. Every knowledge about the data communications protocol was gained by experimenting, try and error and by looking at the data stream. I especially have not decompiled any part of the Yamaha software application nor of the THR's firmware. Described protocol behavior can be totally wrong. I only describe results from (good) guesses and succeeded experiments - not facts!

I can not ensure, that sending messages in the suggested way will not damage the THRII. As well I can not guarantee, that the hardware, I describe will not damage your THRII device or it's power supply or even your computer.

Project goals

The reason for analyzing the protocol was, that I wanted to construct a pedalboard for the THR30II to be able to switch more than the integrated 5 presets, that are stored in the amp and reachable by tapping tiny push buttons.
For a guitarist, settings must be switchable by foot – and 5 sounds are not enough.
For the classical THR family there are 3rd party solutions like the "patchbox" or Mathis Rosenhauer's THR_footswitch or Mehlbrandt's variant of this switch with a very nice 3D printed housing. A commercial device called Patchbox68 was available as well for a period of time.
All these solutions simply counted up to 100 patches with an up and a down button.
My goal was a more musical approach with a button for SOLO and buttons to reach the presets ordered in groups. For example you can be prepared for 10 songs with 5 different presets and 5 solos each.
And I wanted a better display showing names for the patches – who wants to memorize 100 patch numbers or even try them out while having a gig?
Another objective was to get the pedalboard done without an external power supply. I don't like batteries, especially when the are drained and no spare in reach.
The ideal was to tap the rechargeable battery of the THRII to be able to play even without the THRII's power supply.
As I stated above, I wanted to have a better display with the pedalboard. If it is built in anyway, why not use it for more than showing patch names? I decided to show a live bar-chart of the parameters. With some experience you can see the nature of a patch from this chart.

State of the project

After a longer period of software development with the prototype, I now have almost finished the real device. In fact, only the labels for the front panel knobs are missing.

My goals have been reached, except for co-using THRII's rechargeable battery. There simply is no way to tap it from the outside. I have tried almost everything: The USB-jack is powerless, power jack allows no reverse transient, USB data lines are far to weak to harvest power from them, same for the phones jack. Perhaps you could use power from the guitar input jack, because it can charge the G10T. But you can not occupy the jack of course

Midi System Exclusive Message delimiters:

```
f0 …        Message Bytes       … f7
```

Inquiry (starts with general Midi Message PC => THRII)

```
PC:    F0 7E 7F 06 01 F7 (UNIVERSAL_SYSEX Identity Request, non real time)

Depending on the THRII's version and firmware it replies:
THR:   F0 7E 7F 06 02 00 01 0c 24 00 02 00 67 00 2A 01 F7
       (UNIVERSAL_SYSEX Identity Reply, non real time):
       F0 7E KK 06 02 MA MB MC F1 F2 M1 M2 V1 V2 V3 V4 F7
       KK : channel (7f=broadcast)
       MA, MB, MC = Manufacturer (here  00 01 0c = Line6)
       F1, F2 = Family Code (here:  0x0024)
       M1, M2 = Model Number (here: 0x0002)
       V1, V2, V3, V4 = Version Number (here: 1 42 0 'g' with 2A16=4210 and 6716='g')
Depending on the THRII's firmware it continues with two 00-terminated UTF-8
strings in one message:

THR:   f0 00 01 0c 24 02 7e 7f 06 02 4c 36 49 6d 61 67 65 54 79 70 65 3a 6d 61
       69 6e 00 4c 36 49 6d 61 67 65 56 65 72 73 69 6f 6e 3a 31 2e 33 2e 30 2e
       30 2e 63 00 f7
       (L6ImageType:main L6ImageVersion:1.3.0.0.c)
or
THR:   f0 00 01 0c 24 02 7e 7f 06 02 4c 36 49 6d 61 67 65 54 79 70 65 3a 6d 61
       69 6e 00 4c 36 49 6d 61 67 65 56 65 72 73 69 6f 6e 3a 31 2e 34 2e 30 2e
       30 2e 61 00 f7
       (L6ImageType:main L6ImageVersion:1.4.0.0.a)
or
THR:   f0 00 01 0c 24 02 7e 7f 06 02 4c 36 49 6d 61 67 65 54 79 70 65 3a 6d 61
       69 6e 00 4c 36 49 6d 61 67 65 56 65 72 73 69 6f 6e 3a 31 2e 34 2e 32 2e
       30 2e 67 00 f7
       (L6ImageType:main L6ImageVersion:1.4.2.0.g)

or
THR:   f0 00 01 0c 24 02 7e 7f 06 02 4c 36 49 6d 61 67 65 54 79 70 65 3a 6d 61
       69 6e 00 4c 36 49 6d 61 67 65 56 65 72 73 69 6f 6e 3a 31 2e 34 2e 32 2e
       30 2e 67 00 f7
       (L6ImageType:main L6ImageVersion:1.4.3.0.b)
```

Bitbucketing/De-Bitbucketing:

Inside a MIDI-System-Exclusive message only byte values 00...$7F_{16}$ are allowed (7Bit). Raw data to send, however, contains the full range 00..FF_{16} (8 Bit). So before inserting data into the SysEx-frame it has to be ***coded***. Line6 / Yamaha use a certain "***Bitbucketing***" algorithm for that.

7 Bytes of 8-bit raw data are sent out as 8 Bytes coded data with all highest bits zeroed.

```
Raw Data:                 1        2        3        4        5        6        7       Byte numbers
                   76543210 76543210 76543210 76543210 76543210 76543210 76543210      Bit numbers
                   XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX      Raw data bits

Bitbucket      7       6        5        4        3        2        1        0        Bit numbers
               0       X        X        X        X        X        X        X        7 high bits packed in one Byte

Coded Data: Bitbucket     1        2        3        4        5        6        7       Byte numbers
            76543210 76543210 76543210 76543210 76543210 76543210 76543210 76543210    Bit numbers
            0XXXXXXX 0XXXXXXX 0XXXXXXX 0XXXXXXX 0XXXXXXX 0XXXXXXX 0XXXXXXX 0XXXXXXX
```

In ***Bitbucket***-coded data the most significant bit (Bit7) is always 0.

To send out a SysEx-Message we have to collect the 7 highest bits of 7 consecutive raw data bytes and send them in front as the Bitbucket. For simplification only complete 8-Byte-groups of bitbucketed data are sent.
If we receive a Bitbucket-coded message we have to put each bit of the Bitbucket back to the highest bit of it's corresponding data byte. The Bitbucket itself vanishs of course leaving the 7 decoded raw bytes.

```
Coded Data:
       Bitbucket          1        2        3        4        5        6        7       Byte numbers
    7 6 5 4 3 2 1 0 76543210 76543210 76543210 76543210 76543210 76543210 76543210      Bit numbers
    0 X X X X X X X 0XXXXXXX 0XXXXXXX 0XXXXXXX 0XXXXXXX 0XXXXXXX 0XXXXXXX 0XXXXXXX

Decoded Data:
                          1        2        3        4        5        6        7       Byte numbers
                   76543210 76543210 76543210 76543210 76543210 76543210 76543210      Bit numbers
                   XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX      Raw data bits
```

Example:

f0 00 01 0c 24 02 4d 00 0f 00 00 0f 78 7f 7f 7f 7f 55 01 00 03 00 04 00 00 00
16 7f 40 19 3e ~~00 00 00 00 00~~ f7

The Bitbuckets and the affected data bits are colored orange here.
1st Bitbucket is 78_{16}=01111000_2 So he have to set the top bits of the following 4 data bytes to '1'.
2nd Bitbucket is 03_{16}=00000011_2 So he have to set the top bits of the last 2 data bytes of the group to '1'.
3rd Bitbucket is 40_{16}=01000000_2 So he have to set the top bit of the first data byte of the group to '1'.
As you see, the 3rd Bitbucket group is complete in the message frame. But the last bytes 5 Bytes can be ignored, because the last valid byte index is $0F_{16}$= 16_{10}. So the "3e" is the last valid byte in the frame. Keep in mind that the Bitbuckets are ***not*** counted with this index.

Resulting decoded frame (stripped f0..f7 delimiters and the invalid bytes):

```
00 01 0c 24 02 4d 00 0f 00 00 0f ff ff ff ff 55 01 00 00 04 00 00 00 96 ff 99 3e
|------------------------------| 1st longword 2nd longword 3rd longword 4th longword
                                 0xffffffff   0x00000155   0x00000004   0x3e99ff96
```

THRII formatted System Exclusive Message from PC to THR:

```
f0 00 01 0c 24 02 4d 00 01 00 00 07 00 04 00 00 00 04 00 00 00 00 00 00 00 00 00 00 f7
   Line6 THRII Model 01 CNT    L2 L1 BB OP/UKey     FB/PKey  BB    Type        Val
                  A/B    PC0*16+7   |-------------------|  |-------------------|
                            =7            Data 1st BB-group     Data 2nd BB-group
                        Index:      0  1  2  3  4  5  6     7  8  9  A  B  C  D
```

```
THRII Model:
00 24 00 00: THR10II          = 2359296₁₀ (for "device" in thrl6p-file)
00 24 00 01: THR10IIWireless  = 2359297₁₀ ("device" in thrl6p-file)
00 24 00 02: THR30IIWireless  = 2359298₁₀ ("device" in thrl6p-file)
00 24 00 03: THR30IIAcousticWireless = 2359299₁₀ ("device" in thral6p-file)

In early firmware versions, messages from PC to THR were sent with 22 instead of 24.

4d: normally always the same value here (except: 7a to trigger a firmware update and
in conjunction with the G10T)

A/B: A kind of command grouping. System is not clear
     A (00) seems to be more local/actual/command
     B (01) seems to be more global/extended/request

CNT: A frame counter (separate counters for A and for B commands)

L2:  Index of last valid data byte in this frame "div" 16
L1:  Index of last valid data byte in this frame "mod" 16
PC:  Frame #0 of a payload series
     (in traffic to   THR all frames of a payload series have the same current number,
      in traffic from THR it is incremented)
BB:   "Bitbucket" for the following septet of data bytes
     It contains all 7 MSBs combined in one byte (MSB = 0)
     (see: Bitbucketing / De-Bitbucketing)
```

Data is normally sent as 32-bit values (Little Endian). So raw data contains several groups of 4 bytes each (before bitbucketing). The value $268_{10} = 010C_{16}$ would be sent as 0c 01 00 00

```
OP:  Opcode (here: Header for a command)

FB:  Number of data bytes in following frame(s) (here: 4 Bytes in following frame)
```

Some commands are just one single frame stand alone (FB=0). Depending on the opcode OP it may contain only the OP or in addition a Value or a Type and a Value.

Other commands contain a **header** frame (FB≠0) and a following **body** frame.
In body frames instead of OP and FB there are UKey, PKey, Type and Value instead. In some cases, the body frame only contains a Value.

Data types for messages from PC to THRII:

04 = Integer/Floating Point Single (4 Byte / 32Bit)
03 = Boolean (4 Byte: 00 00 00 00 … 00 00 00 01)
02 = Enumeration (4 Byte: 00 00 00 00 …) (Tables)
01 = Single byte (8 Bit) (?) or String(?)

Most parameter values given as 4-Byte are in fact floating point values in IEEE-754 format. Almost all the parameters are stored as [0.0…1.0] and must be scaled to [0.0…100.0].

But "Gate Threshold" is stored as [-9.6…0.0] and should be scaled to [-96dB ... 0dB]. In ".thrl6p"-files they use it this way. But the THR-Remote software shows this parameter scaled to [0.0…100.0].

Opcodes for messages from PC to THRII:

There is a lot of uncertainty with the opcodes, a lot of guessing here!

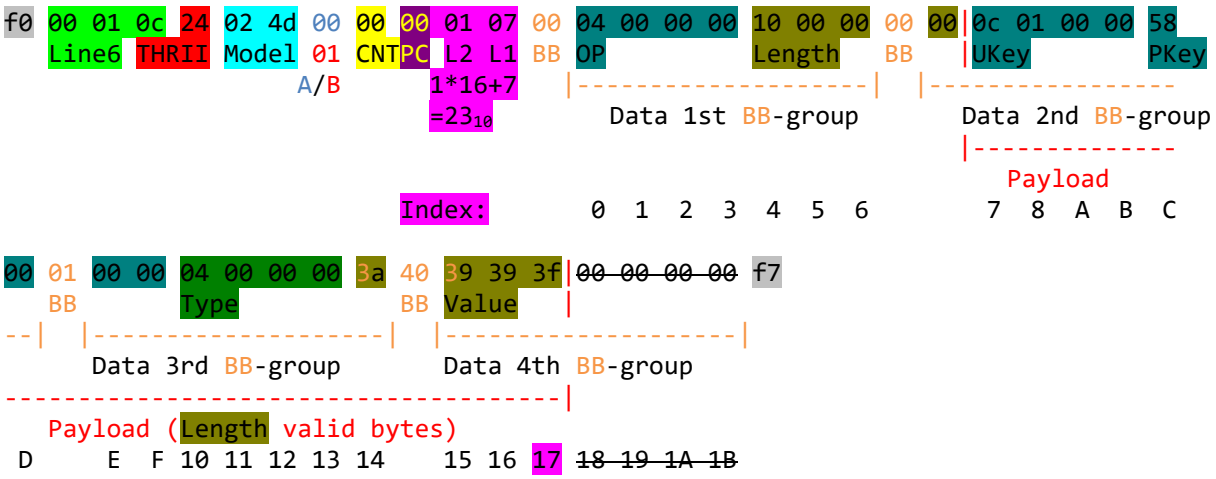
01 = Short System Question (Firmware version. Used as A- and as B-command)
02 = Short System Question (Length of symbol table in bytes. Used as A- and as B-command)
03 = A-command: Short System Question (Symbol table)
B-command:
04 = A-command: Activate Midi-Interface (send 4-Byte magic key in following frame)
B-command:
05 = Short System Question unknown purpose (results in 128 for A-command and in 16 for B-command)
perhaps count avail. units?
06 = A-command: Short System Question (available unit keys)
B-command: Request User patch name download (0-based number as Int-parameter. in same frame)
If no parameter is given it results in a 9 Bytes message (empty string).

07 = A-command: Ask Unit-Type (UKey given as Int-Parameter in following frame). Returns key for the Type.
B-command:
08 = A-command: Set global parameter (e.g. select AMP-model, effect-type)
B-command:
09 = A-command: Ask global parameter (UKey and PKey in following 8-Byte frame, e.g. Tuner enable State or Master / Gain)
B-command:
0A = A-command: Set parameter (e.g. Gain-value or Effect-on/off)
B-command:
0B = A-command:
B-command:
0C = A-command: Status Request (results in "Ack"-Message and "Ready"-Message)
B-command: Request user setting download (settings number as a par. with FFFFFFFF = actual settings)
(see "Download patch")
0D = A-command: System Question (see "Codes for System Questions")
B-command: Upload data block to THRII (e.g. set patch name or upload whole patch)
(see "Upload patch")
0E = A-command: System Setting (see "Codes for System Questions").
Code, type and value follows in 12-Byte body.
B-command: Select user setting (number as 0-based Int-parameter in same frame)
0F = A-command: Short System Question ("Have user settings changed?")
B-command: Results in 37-Byte Answer (unknown purpose)

Codes for System Questions (opcode 0D) and System Settings (opcode 0E) in messages from PC to THRII:

00 = "Number of actual user setting?" (even if it is changed - actual setting is based on it!)
(Read Only!)
01 = "Was user setting changed?"
(Read Only!)
02 = "Is Front-LED on?"
03 = "Wireless channel mode?"
04 = "Wireless channel?"
05 = "??" -> seems to always results in 0
06 = "Is extended Stereo on?" (result 00 for Display "#")
(Read Only!)
07 = "Audio Streaming EQ type?"
08 = "Guitar/Audio-Volume to Line-Out?"
09 = "USB-Output Volume?"
0A = "??" -> seems to always result in 0
0B = "G10T plugged in?"
(Read Only?)
0C = "Battery level?"
(Read Only!)
0D = "Guitar DI Mode?" (Record Dry)
0E = "Speaker Tuner Mode?" (Open or Focused)
0F = "Eco Recharge?"

THRII formatted System Exclusive Message from THR to PC:



THRII Model:
00 24 00 00: THR10II = 2359296_{10} (for "device" in thrl6p-file)
00 24 00 01: THR10IIWireless = 2359297_{10} ("device" in thrl6p-file)
00 24 00 02: THR30IIWireless = 2359298_{10} ("device" in thrl6p-file)
00 24 00 03: THR30IIAcousticWireless = 2359299_{10} ("device" in thral6p-file)

4d: normally always the same value here. (except: 7a in conjunction with the G10T)

A/B: A kind of command grouping. Underlying system is not clear
A (00) seems to be more local/actual
B (01) seems to be more global/extended

CNT: A frame counter (separate counters for A and for B frames)

L2: Index of last valid data byte in this frame "div" 16
L1: Index of last valid data byte in this frame "mod" 16
PC: Frame #0 of a payload series
(in traffic ***to*** THR all frames of a payload series have the same current number,
in traffic ***from*** THR it is incremented)
BB: "Bitbucket" for the following septet of data bytes
It contains all 7 MSBs combined in one byte (MSB = 0)
(see: Bitbucketing / De-Bitbucketing)

Data is normally sent as 32-bit values (Little Endian). So raw data contains several groups of 4 bytes each (before bitbucketing). The value 268_{10} = $010C_{16}$ would be sent as 0c 01 00 00

OP: Opcode (here: Header for MIDI command)

Length: Number of payload bytes in this frame(s) (here: 10_{16}=16_{10} Bytes in this frame)

UKey: A key for the affected THRII-Unit (see "Symbol Table")

PKey: A key for the affected parameter (see "Symbol Table"). For some opcodes PKey is replaced by Value

Type: Data Type/number of data bytes. Normally this is 04. For some opcodes Type is not sent at all.

Value: The value for the parameter. For some opcodes Value is not sent in this place

Types for messages from THRII to PC:

Different from Message from PC to THRII in messages ***from*** THRII to PC different data types are not consistently used.
The "Type" field, is in fact more a "number of data bytes" field than a Type field. But as it is in the same position as the Type field in messages ***to*** the THRII, I stayed with the name "Type".
You will mostly see a `04000000` in this position, at least for parameter change messages (opcode `04000000`).

If an on/off value is sent, Float 0.0 (00000000_{16}) and 1.0 ($3F800000_{16}$) are used as the value, meaning "off" or "on". This is used to switch effect units on or off.
In case of the enumeration constants for "Cabinet", the Value is the Float for 0.0 to 16.0.

For messages to change the Amp-Simulation or the effect unit type a different opcode is used (`03000000` instead of `04000000` for parameter change) and the enumeration values come without a Type field, and they are unique keys from the symbol table.

For messages with opcode `02000000`, Type `02000000` seem to exist, however. In this case two values, that are clear enumerations are preceded each by this Type. See "Reacting to manually changed User Setting on THR30II".

In answer messages (opcode `01000000`) you will also see the Boolean type `03000000`.

`04` = Integer/Floating Point Single (4 Byte / 32Bit)
`03` = Boolean (4 Byte: `00 00 00 00` … `00 00 00 01`)
`02` = Enumeration (4 Byte: `00 00 00 00` …)
[`01` = Single byte (8 Bit) (?) or String(?)]

Opcodes for messages from THRII to PC:

There is a lot of uncertainty with the opcodes, a lot of guessing here!
`01` = Answer
`02` = User setting dump report
`03` = Unit-type change report
`04` = Parameter change report
`05` = ?
`06` = Status message

Acknowledge: (confirm parameter change. Frame from THRII to PC)

If the PC has send a parameter change message to THRII, an acknowledge message is replied by the THRII.

THR: Acknowledge (OK)

```
f0 00 01 0c 24 02 4d 00 57 00 00 0b 00 01 00 00 00 04 00 00 00 00|00 00 00 00|00 00 f7
                                                                  = 0x00000000 (=0)
```

Strictly speaking, this is simply an answer message (opcode 01) with a single parameter with 04 bytes length and a value of 00000000_{16}. But, when you consider, that - if wrong messages are sent from PC to THRII -, a similar reply message but with a value of $FFFFFFFF_{16}$ occures, then it is obvious, that this message has got the purpose of acknowledging / not acknowledging.

THR: Not acknowledge (error)

```
f0 00 01 0c 24 02 4d 00 57 00 00 0b 00 01 00 00 00 04 00 00 3c 00|7f 7f 7f 7f|00 00 f7
                                                                  = 0xFFFFFFFF (=-1)
```

"Not Ack." comes, if non existing units / parameters are addressed. Out of range values, however, are simply limited to their range and confirmed with an "Ack." anyway.
"Ack. " / "not ack." exists as well for B-commands, not only for A-commands. In the opposite direction, the PC (THR-Remote) does not send "ack." / "not ack." messages to the THRII.

Ready-Message:

Before manual parameter-changes on THRII are reported with opcode 03 or 04 -messages, a "ready" frame from THRII to PC occurs. The parameter-change frame follows. After the last parameter change this "ready" frame occurs again:

THR: Frame with opcode 06 and 12 data bytes (3 parameters) inside the same frame.

```
f0 00 01 0c 24 02 4d 00 2a 00 01 03 00 06 00 00 00 0c 00 00 00 00|01 00 00 00 02 00 00
00 00 01 00 00 00|00 f7
```

The meaning of the three parameters is not clear – the never seem to change.
After a "not ack." frame, no "Ready" is sent.

Activating MIDI-communication (starts with protocol message PC => THRII)

Just powered up, the THRII does not react to SysEx commands except the Inquiry message.
After Inquiry the PC must send a message with opcode 04 followed by a magic key to activate the regular MIDI communication.
Because the magic key is depending on the firmware version we can ask for the firmware version before sending the magic key (perhaps it is possible to calculate the fitting magic key from the firmware version).
Be sure to adapt the magic key after a THRII firmware update!

```
PC: (stand alone message 00 bytes follow) "Short System Question 01 (firmware version)"
f0 00 01 0c 24 02 4d 00 00 00 00 07 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f7

THR: (answer 04 bytes)                                                  0x01420067=1.42.0g
f0 00 01 0c 24 02 4d 00 02 00 00 0b 00 01 00 00 00 04 00 00 00 00|67 00 42 01|00 00 f7
```

Now we can pick up the right magic key for this firmware version and activate MIDI with it.

```
PC: (header with Opcode 04; 04 bytes follow in body)
f0 00 01 0c 24 02 4d 00 01 00 00 07 00 04 00 00 00 04 00 00 00 00 00 00 00 00 00 00 f7

PC: (body with magic key as the payload)
f0 00 01 0c 24 02 4d 00 02 00 00 03 28 72 4d 54 5d 00 00 00 f7
                                     0xDD54CD72 (magic key for firmware 1.42.0g)
```

Firmware	Magic Key	Magic Key Bitbucket-coded
1.30.0 c	0x686FBEEB	60 6b 3e 6f 68 ~~00 00 00~~
1.31.0 k	0x9809EB24	28 24 6b 09 18 ~~00 00 00~~
1.40.0 a	0x7986615C	10 5c 61 06 79 ~~00 00 00~~
1.42.0 g	0xDD54CD72	28 72 4d 54 5d ~~00 00 00~~
1.43.0 b (unchanged)	0xDD54CD72	28 72 4d 54 5d ~~00 00 00~~

The THRII responds with an acknowledge message (Opcode 01, 04 bytes data 00 00 00 00).

THR: (Acknowledge)
f0 00 01 0c 24 02 4d 00 01 00 00 0b 00 01 00 00 00 04 00 00 00 00|00 00 00 00|~~00 00~~ f7

When activation is successfully completed, THRII sends out MIDI-SysEx messages, whenever you turn knobs on the device. As well it will react to correct parameter change commands now.
This activation will last until the THRII is powered down. It seems not to do any harm, if you repeat activation several times until then. But you will not need to.
Do not send any commands to the THRII before successful MIDI activation- or you get stuck!
The classic THR-family used to send out a kind of heartbeat-message, so that the remote was able to proof if connection was lost. This heartbeat sometimes was a problem if MIDI-input buffer was not read out regularly. THRII does not do that. But Remote program can proof connectivity by checking acknowledge and reply messages.

Synchronization process

The remote program first gathers some information about the connected device.
1) Short System Question for the firmware version – (again) but this time as a B-command (don't know why!)
f0 00 01 0c 24 02 4d 01 00 00 00 07 00 01 00 00 00 00 00 00 00 00 ~~00 00 00 00 00 00~~ f7
THRII answers to this Short System Question Firmware-Version (length = 4 Bytes btw.|…|)
THR:
f0 00 01 0c 24 02 4d 01 00 00 00 0b 00 01 00 00 00 04 00 00 00 00|67 00 42 01|~~00 00~~ f7
Firmware 1.42.0g
2) Short System Question (unknown purpose)
PC:
f0 00 01 0c 24 02 4d 01 01 00 00 07 00 05 00 00 00 00 00 00 00 00 ~~00 00 00 00 00 00~~ f7
THRII answers with a value of $80_{16} = 128_{10}$. I don't know what this value is for. Everything works fine without knowing this value. Perhaps this value differs for THR10II but for THR30II it is always the same value and not depending on the firmware version.
THR:
f0 00 01 0c 24 02 4d 01 01 00 00 0b 00 01 00 00 00 04 00 00 20 00|00 00 00 00|~~00 00~~ f7
0x00000080=???
3) Short System Question ("Have User-Settings changed?")
PC:
f0 00 01 0c 22 02 4d 00 03 00 00 07 00 0f 00 00 00 00 00 00 00 00 ~~00 00 00 00 00 00~~ f7
THRII answers with a value of 00 (=not changed) or 01 (=changed). With this information, the Remote software can show this with red or beige color for the settings name (like the vanishing settings number on the THRII's LED-display).
THR:
f0 00 01 0c 24 02 4d 00 02 00 00 08 00 01 00 00 00 01 00 00 00 00|01|~~00 00 00 00 00~~ f7

changed

4) Request User Settings (B-Command with Opcode 0c download of user setting with number as parameter)

```
f0 00 01 0c 24 02 4d 01 02 00 00 0b 00 0c 00 00 00 04 00 00 3c 00 7f 7f 7f 7f 00 00 f7
                                                                  FFFFFFFF16 (=actual values)
```

THRII responds with a **dump message series** containing all the actual settings.
(see "Settings dumps")
After the dump the THRII will finish with a "User Settings Dump Report" (Opcode 02):
THR:

```
f0 00 01 0c 24 02 4d 00 03 00 01 07 00 02 00 00 00 10 00 00 03 00|02 00 00 00 7f
   7f 60 7f 7f 02 00 00 00 01 00 00 00 00|00 00 00 00 f7
```

This message has a payload of 10_{16} = 16_{10} Bytes (|…|).
First value 02 means, that this setting is derived from user setting #3 (user setting #1 would be a 00).
$FFFFFFFF_{16}$ shows us, that it is the ***actual*** setting, that was dumped.
The 02 declares, that the following value is an enumeration type. The value 01 stands for "downloaded to PC" instead of 00= uploaded to THRII.

5) System Question "Number of actual user setting"

```
PC: (Header, Opcode 0d=System-Question, 04 Bytes follow in body frame)
f0 00 01 0c 22 02 4d 00 04 00 00 07 00 0d 00 00 00 04 00 00 00 00 00 00 00 00 00 00 f7

PC: (Body, Code 00=Number of actual User Setting)
f0 00 01 0c 22 02 4d 00 05 00 00 03 00 00 00 00 00 00 00 00 00 f7
```

THRII answers with the 0-based number (Opcode 01 = Answer, $0c_{16}$=12_{10} Bytes payload btw.|…|)

```
THR:(Answer to System-Question)                                                  enum-Type
f0 00 01 0c 24 02 4d 00 04 00 01 03 00 01 00 00 00 0c 00 00 00 00|00 00 00 00 02 00
   00 00 00 04 00 00 00|00 f7
            Setting #5 is active
```

The expression "active" setting can be misleading here. If you press one of the push buttons "User Memory 1…5" on the THRII, a complete setting becomes active, and the number is lit on the LED. If you afterwards touch any knob altering a parameter even slightly, the number vanishes, showing that the selected setting was altered. But still this actual setting somehow belongs to the user memory setting, it was derived from.
So even if there is no number lit on the LED anymore, the last user setting is still the active one.
The first value in the payload section between |…| is a 00. It is not totally clear, what that stands for. It could be the status and would (I guess) mean "OK" in this case.

6) Ask global Parameter "Tuner enabled?"
If you start the THR Remote program while the THRII is in Tuner mode (shown on LED with ▸●◂) you'll get the message "*The application is unavailable while the amp is in tuner mode*". Perhaps it would be better if the application simply turned off tuner mode in this case. But it does not and perhaps it is not able to do this(?). The following frames retrieve this information, so that THR Remote can react with the message above.

```
PC: (Header, Opcode 09=Ask global parameter, 08 Bytes follow in body frame)
f0 00 01 0c 24 02 4d 00 06 00 00 07 00 09 00 00 00 08 00 00 00 00 00 00 00 00 00 00 f7
PC: (Body, UKey=FFFFFFFF16, PKey=0000014f16)                    "TunerEnable"
f0 00 01 0c 24 02 4d 00 07 00 00 07 78 7f 7f 7f 7f 4f 01 00 00 00 00 00 00 00 00 00 f7

THR: (Opcode=01=Answer, 0c16=12 Bytes, OK, bool, not enabled)
f0 00 01 0c 24 02 4d 00 05 00 01 03 00 01 00 00 00 0c 00 00 00 00|00 00 00 00 03 00 00 00
00 00 00 00 00|00 f7
```

7) Read out the name strings for all User Memory settings.
PC: Stand alone message, B-Command Opcode 06 (Download Name of User-Setting#1 00)

```
f0 00 01 0c 24 02 4d 01 03 00 00 0b 00 06 00 00 00 04 00 00 00 00 00 00 00 00 00 00 f7
```

THR: Answer to Request(Opcode 01, data length = 15_{16}=21_{10} Bytes betw. |…|, $0d_{16}$=13_{10} characters incl. 00)

```
f0 00 01 0c 24 02 4d 01 07 00 01 0c 00 01 00 00 00 15 00 00 00 00|00 00 00 00 0d 00 00
00 00|54 61 6b 65 20 00 69 74 20 65 61 73 79 00 00||00 00 00 00 00 00 f7
                T  a  k  e  ␣     i  t  ␣  e  a  s  y
```

This must be repeated for User-Setting #2, #3, #4 and #5

8) Ask global Parameters "Guitar Volume" and "Audio Volume"

In actual firmware the state of Guitar and Audio Volume knobs is monitored and adjustable by the remote software, acting now like Gain, Master and Tone knobs. Before, their values were as well not transmitted by SysEx, if changed.

As all parameter values must be known by the remote software, now Guitar- and Audio Volume must be included in the syncing process. But these values are not part of the "actual Settings" requested and delivered above in 4), they must be requested separately. If you ask, why they are not part of the "actual Settings", I'd say this is because they must stay unchanged if a "User Memory Settings" push button is pressed or another patch is loaded with the Remote software.

```
PC:                Opcode 09= Ask global parameter   08 bytes in following frame
f0 00 01 0c 24 02 4d 00 08 00 00 07 00 09 00 00 00 08 00 00 00 00 00 00 00 00 00 00 f7
                                        FFFFFFFF=global  5501=GuitarVolume
f0 00 01 0c 24 02 4d 00 09 00 00 07 78 7f 7f 7f 7f 55 01 00 00 00 00 00 00 00 00 00 f7

THR:   Answer to question (GuitarVolume)
f0 00 01 0c 24 02 4d 00 06 00 01 03 00 01 00 00 00 0c 00 00 00 00|00 00 00 00 04 00 18
00 00 6a 69 69 3f|00 f7

PC:           09= Ask global parameter  08 bytes in following frame
f0 00 01 0c 24 02 4d 00 0a 00 00 07 00 09 00 00 00 08 00 00 00 00 00 00 00 00 00 00 f7
                                      FFFFFFFF=global  4b01=AudioVolume
f0 00 01 0c 24 02 4d 00 0b 00 00 07 78 7f 7f 7f 7f 4b 01 00 00 00 00 00 00 00 00 00 f7

THR:   Answer to question (AudioVolume)
f0 00 01 0c 24 02 4d 00 07 00 01 03 00 01 00 00 00 0c 00 00 00 00|00 00 00 00 04 00 1c
00 00 41 40 40 3e|00 f7
```

9) Ask global parameters "G10T plugged in?", "Front LED on?" and "Speaker Tuner state?"
In actual firmware the state of Front LEDs can be switched on/off again as it was possible already for the classical THR series. Thus, the Remote should be aware of the actual state.
Another global information that the Remote software will need, is if the G10T wireless guitar transmitter is plugged in. If yes, the Remote will ask for it's firmware version, to do an update if necessary (G10T updating feature was added, when cases of burning by overloading got known, as far as I know).
In my Desktop program I use this information for showing plug in state in the graphical user interface.
Finally the relatively new global parameter "Speaker Tuner", that can be "Open" or "Focus" should be asked.

```
PC:          0d=SystemQuestion   04 bytes in following frame
f0 00 01 0c 24 02 4d 00 0c 00 00 07 00 0d 00 00 00 04 00 00 00 00 00 00 00 00 00 00 f7
                                                  Code 0b= G10T plugged in?
f0 00 01 0c 24 02 4d 00 0d 00 00 03 00 0b 00 00 00 00 00 00 f7

THR:       Answer    12Byte, OK,enum, off (not plugged in)
f0 00 01 0c 24 02 4d 00 08 00 01 03 00 01 00 00 00 0c 00 00 00 00|00 00 00 00 02 00 00
00 00 00 00 00 00 00 00|00 f7


PC: Header                  Opcode 0D for Syst.Read    04 bytes in following frame
f0 00 01 0c 24 02 4d 00 0f 00 00 07 00 0d 00 00 00 04 00 00 00 00 00 00 00 00 00 00 f7

                    code "Front-LED on" = 02
f0 00 01 0c 24 02 4d 00 10 00 00 03 00 02 00 00 00 00 00 00 f7

THR:                                          01=Answer   0c=12 Bytes    00=OK        02=enum
f0 00 01 0c 24 02 4d 00 09 00 01 03 00 01 00 00 00 0c 00 00 00 00|00 00 00 00 02 00 00
      01=active
00 00 01 00 00 00 00 00|00 f7


PC:  Header 0D for Syst.Read Speaker-Tuner state
f0 00 01 0c 24 02 4d 00 11 00 00 07 00 0d 00 00 00 04 00 00 00 00 00 00 00 00 00 00 f7

     Syst. Read Speaker-Tuner Value code 0e=Speaker-Tuner
f0 00 01 0c 24 02 4d 00 12 00 00 03 00 0e 00 00 00 00 00 00 f7

THR:                                          01=Answer   0c=12 Bytes    00=OK        02=enum
f0 00 01 0c 24 02 4d 00 09 00 01 03 00 01 00 00 00 0c 00 00 00 00|00 00 00 00 02 00 00
      01=Focus
00 00 01 00 00 00 00 00|00 f7
```

Settings Dumps

The most interesting part of the SysEx protocol is the possibility to send and receive a complete set of parameter settings and not one single parameter.
Let us call a complete set of parameters a "**patch**". A patch is the same than a "User Memory Setting" invoked by pressing one of the 5 push buttons on the THRII.

A **patch** contains a lot of data, because it consists of several parameters for several effect units. In addition there is a patch name (up to 63 characters long). And keep in mind, that the parameters are stored in 4 bytes each, even if it is a simple boolean Unit on/off value.
The classic THR10 had a more compact data format. On a THR10 a patch had a fixed size of 273 bytes.
On THRII-family the patch size is variable but about 600 to 1200 bytes. For this reason, it does not fit in a single SysEx frame. For THRII no SysEx frame exceeds 310 bytes (including header + 256 valid bytes + bitbuckets and delimiters). So, the patch data comes in a series of about 3 to 5 frames.
Frames belonging to a payload series are counted with PC in the frame's "last index field" 00 00 07.
Normally the frames of a payload series are of the same length with the last one perhaps shorter.
In patch dumps from THRII to PC you will normally see 00 0f 0f in the first frames and e.g. 04 06 01 in the last frame.
In patch dumps from PC to THRII you will normally see 00 0d 01 in the first frames and e.g. 02 0a 05 in the last frame.

Data types in a patch dump:

-Bool: 00 00 01 00 (Value follows stored in 4 bytes)
-Enum: 00 00 02 00 (Value follows stored in 4 bytes)
-Int/Float: 00 00 03 00 (Value follows stored in 4 bytes)
-String: 00 00 04 00 Number of bytes (including 00) follows as a 4-byte-Int, then a chain in UTF-8 followed by 00
-Block: 00 00 05 00 (18 bytes to follow)
00 00 06 00

Structure of a block:
2 Byte Key + "00 00 05 00" + "00 80 07 00"+ 2 Byte Info + "00 00 06 00" + 00 80 02 00 + 4 Byte num. Param.
Example:
12 01 02 00 00 05 00 00 00 07 20 00 71 00 00 00 06 00 20 00 00 02 00 02 00 00 20 00
De-Bitbucketed:
12 01 00 00 05 00 00 80 07 00 F1 00 00 00 06 00 00 80 02 00 02 00 00 00
UKey DataType Block Pseudo 1 Info (Effect type) Data Type Pseudo 2 Num. Param.
Type "Block Reverb"

Special markers in a patch dump message:

50 53 52 50 PSRP initiates section with patchname field
54 52 54 47 TSTG initiates section with first parameter block

Each parameter block starts with a "pseudo key" 03 00 (Dummy value: 00 80 07 00) and ends (except BlockMIX/GuitarProc) with the "pseudo key" 04 00 (Dummy value: 00 80 00 00)

After last parameter block "pseudo key" 04 00 (Dummy value:00 80 00 00) follows a second time. This seems to be the end of the block MIX/GuitarProcessor.
The complete patch is closed with "pseudo key" 02 00 (Dummy value: 00 80 00 00).

Structure of a patch and appearing tokens:

The patch dump data reflects the structure "data" of a ".thrl6p" patch file – but only parts of it. These ".thrl6p"-files are in JSON format and are structured by curly braces. In the patch dump the structure is built by tokens.

```
00 00 00 80 02 00 { Begin Structure meta

      02 00 00 00 01 00 00 80 02 00 50 53 52 50  Token for structure name "meta"

                                 00 00 00 00 04 00 0d 00 00 00 … Patchname 13 chars
                                 01 00 00 00 02 00 00 00 00 00 tnid
                                 02 00 00 00 02 00 00 00 00 00 ?? unknown
                                 03 00 00 00 03 00 00 00 00 00 ParTempo 110 (min)
02 00 00 80 00 00 } End Structure meta

00 00 00 80 02 00 { Begin Structure data

      01 00 00 00 01 00 00 80 02 00 54 52 54 47  Token for structure name "data"


03 00 00 80 07 00            { Begin Block GuitarProcessor
                               3c 01 00 00 05 00 00 80 07 00 BlockGuitProc
                               08 01 00 00 06 00 00 80 02 00 0b 00 00 00 11 Par.
                               Y2GuitarFlow
                               17 01 00 00 03 00 00 00 00 3f
                               18 01
                                  …
                               27 01
03 00 00 80 07 00              {
                                   09 01 00 00 05 00 00 80 07 00 BlockCompr(FX1)
                                   bd 00 00 00 06 00 00 80 02 00 02 00 00 00 2 Par
                                   RedComp
                                   bf 00
                                   be 00
04 00 00 80 00 00              }, End Block Compressor
03 00 00 80 07 00              {
                                   0c 01 00 00 05 00 00 80 07 00 BlockAmp
                                   88 00 00 00 06 00 00 80 02 00 05 00 00 00 5 Par
                                   CrunchClassic
                                   4f 00
                                   52 00
                                   53 00
                                   50 00
                                   51 00
04 00 00 80 00 00              }, End BlockAmp
03 00 00 80 07 00              {
                                   0e 01 00 00 05 00 00 80 07 00 BlockEffect(FX2)
                                   e6 00 00 00 06 00 00 80 02 00 04 00 00 00 4 Par
                                   StereoSquareChorus
                                   dd 00
                                   d2 00
                                   e3 00
                                   e7 00
```

```
04 00 00 80 00 00                }, End BlockEffect
03 00 00 80 07 00                {
                                     11 01 00 00 05 00 00 80 07 00 BlockEcho(FX3)
                                     eb 00 00 00 06 00 00 80 02 00 04 00 00 00 4 Par
                                     TapeEcho
                                     4f 00
                                     d2 00
                                     6c 00
                                     51 00
04 00 00 80 00 00                }, End BlockEcho
03 00 00 80 07 00                {
                                     14 01 00 00 05 00 00 80 07 00 BlockReverb(FX4)
                                     fe 00 00 00 06 00 00 80 02 00 03 00 00 00 3 Par
                                     ReallyLargeHall
                                     f8 00
                                     79 00
                                     72 00
04 00 00 80 00 00                }  End BlockReverb

04 00 00 80 00 00              } End Unit GuitarProcessor
02 00 00 80 00 00  } End Structure "data"
```

Compressor (FX1) Type
The Compressor-Type is coded inside Block 0901[1] for Compressor-Unit (FX1). (It is fixed to value BD00="RedComp"). In THR-Remote no Compressor type is selectable.
Perhaps type C400="Stomp" could work(?).
It is interesting that a second constant C700="Red Comp" exists (with a space char inside).

Effect (FX2) Type
Effect-Type is coded as a type field in Block 0E01 for the Effect-Unit (FX2). In the example above e600 "StereoSquareChorus" is selected.

Echo (FX3) Type
Echo-Type is coded as a type field in block 1101 for the Echo-Unit (FX3). (Before 1.40.0a it was fixed to value EB00="TapeEcho"). In THR-Remote until 1.40.0a no Echo-Type was selectable.
It is interesting that a second constant F000="Tape Echo" exists (with a space char inside).
Since Firmware 1.40.0a besides type "TapeEcho" an Echo-Type F100="DigitalDelay" exists (called "L6DigitalDelay" in the symbol table). Since then, Echo-Type value is not fixed anymore!

Reverb (FX4) Type
Reverb-Type is coded as a type field in block 1401 for the Reverb-Unit (FX4).
In the example above "Hall" (called "ReallyLargeHall" in the symbol table) is selected.

AMP-Simulation
The AMP-Simulation is coded as a type field in block 0C01 for the Amp-Unit (Main-Controls).
In the example above "CrunchClassic" (called "THR10C_DC30" in the symbol table) is selected.

All the **parameters** are stored as type and value inside the patch dump. In the example above this is only shown for 17 01 00 00 03 00 00 00 80 3f . This is "FX1MixState" (A parameter not adjustable in THR Remote!). This parameter is type Bool and set to 0x3f800000 meaning "On".
All other parameters are left out to increase clarit

[1] All the keys are dependent of the firmware version. Here all keys are for actual 1.42.0g firmware

Parsing incoming patch dumps

To parse the dump data a "state machine" can be used. In the pedalboard's Teensy program I use:

```
enum States { St_none, St_idle, St_structure, St_meta, St_global, St_data, St_unit, St_subunit,
              St_valuesUnit, St_valuesSubunit, St_error };
```

If in state "idle" only entering a "structure" is possible. If in state "structure", states "data" or "meta" can be entered. But because there are only values (the global values "tnid" and "tempo") but no subunits, reaching state "meta" directly proceeds to state "global".

If in state "data", state "unit" can be reached or – if a closing token comes in – state "idle" again.

If in state "unit", states "subunit" or "valuesUnit" can be reached or – if a closing token comes in – state "data" again.

If in state "subunit", state "valuesSubUnit" can be reached or – if a closing token comes in – state "unit" again.

Finally arrived in one of the "value…"-states, sets of type key, unit key and parameter key are extracted until a closing token comes in.

Unexpected incoming tokens bring the state machine to "error" state. A successful walkthrough ends by receiving a closing token while in state "data" bringing back state "idle".

Patch dump from THRII to PC / Download patch

A patch dump from THRII to PC only occurs requested by the PC. The PC sends opcode `0c` together with the required user settings number as a 4 Byte parameter in a B-command frame ($FFFFFFFF_{16}$ is actual settings).

```
PC:
f0 00 01 0c 22 02 4d 01 02 00 00 0b 00 0c 00 00 00 04 00 00 3c 00|7f 7f 7f 7f|00 00 f7
```

THRII answers (opcode `01`) with several B-frames. The first one itself has got a last valid index of FF_{16} (contains 256 valid bytes). With header, bitbuckets, delimiters and some invalid bytes, that complete the last bitbucket group, the total SysEx lenght of the first frame in this case is 310 bytes.

```
THR:  Answer to request, frame #1 (PC=00)
f0 00 01 0c 24 02 4d 01 02 00 0f 0f 00 01 00 00 00 57 04 00 00 00|00 00 00 00 00 00
00 00 00 01 00 00 00 00 01 00 00 00|00 00 00 00 00 02 00 ...
```

In this first frame of the payload series directly behind the opcode `01` the total length of the patch data including 4 leading 32bit values (`00000000`$_{16}$ `00000000`$_{16}$ `00000001`$_{16}$ `00000000`$_{16}$) is given. Here this is `00000457`$_{16}$=1111_{10} bytes. I mark the beginning with | here. The real patch dump data begins 16 valid bytes later marked by | , so that it has got a length of 1095 bytes.
The last frame has a "last valid index field" of `04 05 0e` showing us, that it carries the last 5*16+14 +1= 95 bytes. All together 4*256 + 95 = 1119 valid bytes were sent if we add up all valid index values. Subtracting the 4 leading values, the opcode and the length field (each 4-Bytes) we result in 1119-16-4-4= 1095 bytes as expected. The program must concatenate the valid data and afterwards parse it.

THRII finishes the dump sending a user setting dump report message (Opcode `02`).

Patch dump from PC to THRII / Upload patch

A patch dump coming from PC (or from a microcontroller pedalboard) to the THRII starts with a header frame of command B-type with opcode 0d and 6 parameters.
The patch data itself follows in several body frames.

Note, that the body frame(s) contains the pure patch dump data here, while for patch dumps coming from the THRII the first frame of an answer to a patch request starts with 6 parameter values and dump data inside the same frame.

```
PC:   (Example: Overwrite user patch #3. The header frame)
f0 00 01 0c 22 02 4d 01 0b 00 01 0b 00 0d 00 00 00 6e 02 00 00 00|02 00 00 00 66 02
00 00 00|00 00 00 00 01 00 00 00 00 00 00 00 f7

f0 00 01 0c 22 02 4d 01 0c 00 0d 01 08 00 00 00 00 02 00 02 01 00 00 00 01 00 00 00 ...
```

In the "last valid index" field a frame counter PC is present. The first frame has a 00 here, and in this example, the last one has got a 02, because here 3 body frames are sent.

The **first** parameter after the opcode 0d is the length field. In the example above $026E_{16}=622_{10}$ bytes is the total length of data to follow in this payload series. I have marked the first byte of these 622 bytes with | in the example.

The **second** parameter is the 0-based user settings number where THRII shall store the setting, 02 = User Setting #3 in this case.

In the **third** parameter again a length value is stored. It always shows 8 Bytes less than the total length field. This way it shows the length of the pure patch dump data including its leading three 4-byte parameters (00000000_{16} 00000001_{16} 00000000_{16}).

Note that there are only three parameters (a total of 12 bytes) preceding the real patch dump data for uploading patches while there are four parameters (a total of 16 bytes) for downloading patches. But the values are equal in both cases only for download there is an additional leading 00000000_{16}.

Also note, that for uploading patches the frames are not filled to 256 bytes. Instead, the frames have only 210 bytes instead, making 30 completely filled bitbucket groups. Only in the last frame of the series data may end in before the end of a bitbucket group.

But besides these differences real patch dump data is exactly the same for upload and download.

So, for the structure see section Patch dump from THRII to PC .

Symbol Table

One of the most amazing results while analyzing the SysEx protocol was when I sent a special message and received a payload series of 8197 bytes in total. Because of the big size I thought of that being a complete memory settings dump.
But after looking closer at the data, I could see ASCII data with interesting keywords in the last frames of the dump:

```
R30_StealthvisibletrueTSGainTSToneTSDrive-
28.0THR30_SR1010.51SPKSIM_MesaMK3THR30_Hairy0.39THR30_Blondie0.32-
58THR10_Bass_MesaBassAmp0.52BASSTHR30_FLead0.34SPKSIM_Diezel412THR30_Ca
rmen0.48THR10_Flat_BTHR10_Flat_VTHR10C_BJunior2THR
```

This made me curious. Further analysis showed:
In the data, beginning from a certain offset, I could find 373 strings (separated by 00). And because the first 4-Byte word received in this dump was $00000174_{16} = 372_{10}$ this was not likely by chance.
The next 4-Byte word in the received dump was $00002005_{16} = 8197_{10}$, exactly the size in bytes of the received data.
The data between these two leading words and the offset where the readable key words start, were 4476 Bytes, and – no surprise – could be divided by 373 without a remainder – 12 bytes for each of the 373 keywords.
So, it seemed clear to me, that this data dump had to be a kind of **symbol table**.

Each symbol has an associated data of 12 bytes (3*4 bytes)

Nr.	**Symbol**	**Start offset**	**CRC32**	**Length of symbol**
0000	7BandSpkEq	- 00 00 00 00	CB CB 6E 37	0A 00 00 00
0001	PreGain	- 0B 00 00 00	1C 3F 62 F8	07 00 00 00
0002	PostGain	- 13 00 00 00	E1 74 FD 9D	08 00 00 00
0003	EQ1Type	- 1C 00 00 00	CF E8 64 DD	07 00 00 00
0004	Units	- 24 00 00 00	49 74 B0 E9	05 00 00 00
0005	SpkEQ1Type	- 2A 00 00 00	B4 1D 83 8C	0A 00 00 00
0006	displayName	- 35 00 00 00	98 63 2F 67	0B 00 00 00
0007	EQ1Fc	- 41 00 00 00	25 1F B1 72	05 00 00 00
0008	SpkEQ1Fc	- 47 00 00 00	C6 EC 43 84	08 00 00 00
0009	EQ1Gain	- 50 00 00 00	E6 92 2F 81	07 00 00 00
000A	SpkEQ1Gain	- 58 00 00 00	9D 67 C8 D0	0A 00 00 00
000B	EQ1Q	- 63 00 00 00	44 55 C1 3A	04 00 00 00
000C	SpkEQ1Q	- 68 00 00 00	91 ED 14 71	07 00 00 00
000D	SpkEQPreGain	- 70 00 00 00	95 B6 92 8F	0C 00 00 00
…				
0172	Gain2_dB	- 77 0E 00 00	CA 13 C3 C6	08 00 00 00
0173	Level2	- 80 0E 00 00	29 10 5D 99	06 00 00 00
0174	SHIFT	- 87 0E 00 00	F5 F9 5F 52	05 00 00 00

The first and the third 4-byte values are easy to understand as the starting offset and the length of each symbol.
The second 4-byte value associated to each name is the CRC32 hash of the symbol name and you will probably not need it. But, if you number all the symbols in hexadecimals, the number of a symbol is the same number appearing inside SysEx-messages (e.g., 004C is the symbol "Master" and also appears in SysEx Messages when Master volume is changed). In fact, a lot of the symbols are unit keys and parameter keys or enumeration constants like the amp simulation number keys.
This of course was a game changer in the analysis of the protocol. I also found out, that with each firmware update the symbol table changed a bit and grew a little.

So, I decided to always let my program request a download of the symbol table as soon as possible in the boot up process. This way my program would not depend on fixed assignments of parameters and units to their key values.

Requesting the Symbol Table

To request the download, the PC (or microcontroller pedal board) must send a message with opcode 03 and a 00 in the field for "following bytes in next frame" (a standalone message, not header + body).

PC:

```
f0 00 01 0c 24 02 4d 00 03 00 00 07 00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f7
```

THRII sends out an answer message (opcode 01). After the opcode a length field for the whole payload series is given (here $201d_{16}=8221_{10}$ bytes, beginning at |)
As shown above the two first 4-byte words of the download give us number of symbols (here 0175_{16} = 373_{10}) and the length of the table (here $201d_{16}$ = 8221_{10} bytes). Length of the payload series and length of the symbol table are the same because nothing else than the table is sent. After these two values the symbol table itself starts (marked with | here).

THR: Answer (opcode 01)

```
f0 00 01 0c 24 02 4D 00 2e 00 0f 0f 00 01 00 00 00 1d 20 00 00 00|75 01 00 00 1d
20 01 00 00|00 00 00 00 4b 40 4b 6e 37 0a 00 00 00 00 0b 00 00 00 ...
```

The receiving device (PC or pedal board) now must bring the content of the whole series into a buffer and calculate, where inside this buffer the symbols begin. With variable "vals" being the number of symbols in the table (first word value in the table dump) we program something like:

```
symStart = 8 + 12 * vals; //Where is the start of the symbol names?
```

Now the program can extract strings from this position on to the end of the buffer and put them into a container or array, that will act as a dictionary to translate symbol name into key number, whereas the key number is simply the index of the symbol in the container.

For a microcontroller program handling dynamic data always is a challenge. A general hint is, to use stack storage wherever possible. Nevertheless, I found a balance of programming convenience and avoiding heap storage.

Monitoring parameter changes

Whenever a knob is turned or a switch is operated on the THRII, one or several SysEx messages are sent out (at least, if MIDI interface was activated before).
These parameter change messages must be parsed by the Remote software to hold its internal state synchronized to the THRII. A typical message will be:

```
f0 00 01 0c 24 02 4d 00 00 00 01 07 00 04 00 00 00 10 00 00 00 00|0c 01 00 00 58
                                    OP          Length    BB    |UKey       PKey
00 01 00 00 04 00 00 00 3a 40 39 39 3f|00 00 00 00 f7
   BB       Type          BB Value    |
```

After De-Bitbucketing and cutting behind the last valid index of 23 the frame's data appears as:

```
04 00 00 00 10 00 00 00|0c 01 00 00 58 00 00 00 04 00 00 00 ba b9 39 3f|00 00 00 00
```

The parser decides from the opcode 04, that a parameter was changed on THRII. The length field of 10_{16} = 16_{10} lets it extract 4 words (in single frame messages this field is redundant – the "last valid index" field in the header gives the same information in this case).

Now, the first 4-Byte number is the unit key (UKey). The parser must look it up in the symbol table.
It can expect one of the following units:

Friendly name	Name in symbol table	UKey in firmware 1.42.0g	Bytes in SysEx
Main Controls	Amp	0x0000010C	0C 01 00 00
Compressor	FX1	0x00000109	09 01 00 00
Effect (Modulation)	FX2	0x0000010E	0E 01 00 00
Echo	FX3	0x00000111	11 01 00 00
Reverb	FX4	0x00000114	14 01 00 00
Gate, Mix, Units On/off Cabinet	GuitarProc	0x0000013C	3C 01 00 00
Global Parameters	---	0xFFFFFFFF	7f 7f 7f 7f

In the example above, the Main Controls Unit is concerned, and the parser has to proceed to the parameter key (PKey). It is necessary for the parser to analyze the UKey first because the same key can mean a different parameter. For example, there is a PKey 0x00000054 ("Bass") as well in the Main Controls as in the Echo unit.
Here, the PKey 0x00000058 translates to "Drive", which is the symbol table's expression for the "Gain" knobs value.
As the next step, the parser checks the Type field. With this information it can get the correct interpretation of the data value in the last 4-byte value.

Float Values

For the example above, the Type Field says 0x00000004. This is an Integer value or a Float value in 4 Bytes. In case of "Drive" (and the other main controls) the float value will be in the interval of [0.0 to 1.0] and has to be scaled to [0.0 to 100.0] by multiplying it by 100.

A word about the decoding of a float value coming inside a byte-buffer (array of unsigned char) in Little Endian format, like the 4 bytes ba b9 39 3f in the example. Here, the language "C++" is convenient because it can use pointers and type casting.
If the de-bitbucketed message above is contained in an array "buf":

```
unsigned char buf[]={0xf0, 0x00, 0x01, 0x0c, 0x24, 0x02, 0x4d, 0x00, 0x00, 0x00, 0x01,
   0x07, 0x04, 0x00, 0x00, 0x00, 0x10, 0x00, 0x00, 0x00, 0x0c, 0x01, 0x00, 0x00, 0x58,
   0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0xba, 0xb9, 0x39, 0x3f, 0x00, 0x00, 0x00,
   0x00, 0xf7};
```

Then we could access the first byte of the 4-Byte value we need with buf[32].
But we do not want a single byte, but 4 bytes in series used as a 32Bit value. The 32-Bit value type is called **unsigned long int** or often type-aliased to **uint32_t** in C++.

So, we create the following expression to cast a byte-pointer pointing to the first byte into a **uint32_t** pointer and dereference it:

uint32_t num = *((**uint32_t***) (buf+32));

Now we have to read this value as a **float** variable:

```
float val;
```

Though it would work, I do not use a "classical" C-style casting like:

```
val = *((float*) &num);
```

Instead, I copy the 4 bytes directly into the **float** variable with a `memcpy`:

```
memcpy(&val,&num,4);
```

This is more likely to work in modern C++ implementations, that do not allow this sort of type-punning because they are following strict-aliasing semantics.

After the `memcpy`, the variable "`val`" can be read out as a **`float`** value.
Another approach would use a **`union`**, that parallels a **`float`** and an **unsigned long int** variable.

Of course, these techniques are non-portable because they rely on the correct endianness of your device. But as I wrote the program for Teensy only, this should be guaranteed.

Other Types

A boolean type *should* occure for example, if an effect Unit is switched on/off.
But in fact THRII sends a Float value instead.
This will happen, in case the "EFFECT" or the "ECHO/REV" knob is turned and you leave an effect area.

Example:

```
f0 00 01 0c 24 02 4d 00 20 00 01 07 00 04 00 00 00 10 00 00 00 00|3c 01 00 00
2f 01 00 00 00 04 00 00 00 00 20 00 00 3f|00 00 00 00 f7

0000013c = Unit "GuitarProc"
0000012f = Parameter "FX2Enable" (EFFECT enable) val. 3F800000₁₆ = Float 1.0 = ON
```

An enumeration type *should* occure for example, if an effect type is changed or an amp simulation is set. But in fact THRII uses a special opcode (`03000000` instead of `04000000` for parameter change) and the enumeration values come without a Type field and they are unique keys from the symbol table.

Example:

```
f0 00 01 0c 24 02 4d 00 5e 00 00 0f 00 03 00 00 00 08 00 00 02 00|0c 01 00 00
36 00 00 00 00|00 00 00 00 00 f7

0000010c = "Amp"
000000b6 = "THR10C_BJunior2" = CLEAN-BOUTIQUE
```

Assignment AMP-names to symbol table

AMP-Sim	Modern	Boutique	Classic
CLEAN:	THR30_Carmen	THR10C_BJunior2	THR10C_Deluxe
CRUNCH:	THR30_SR101	THR10C_Mini	THR10C_DC30
LEAD:	THR10_Brit	THR30_Blondie	THR10_Lead
HI GAIN:	THR10X_Brown2	THR30_FLead	THR10_Modern
SPECIAL:	THR30_Stealth	THR10X_South	THR10X_Brown1
BASS	THR30_JKBass2	THR10_Bass_Mesa	THR10_Bass_Eden_Marcus
ACO:	THR10_Aco_Dynamic1	THR10_Aco_Tube1	THR10_Aco_Condenser1
FLAT:	THR10_Flat_V	THR10_Flat_B	THR10_Flat

As well an enumeration Type *should* occure, if a different cabinet is selected.
But in fact a parameter change opcode (04000000) is used with a value with Float Type field (04000000).
The value in this case is Float for 0.0 to 16.0.

Example:

```
f0 00 01 0c 24 02 4d 00 5f 00 01 07 00 04 00 00 00 10 00 00 00 00|3c 01 00 00
07 01 00 00 00 04 00 00 00 00 00 00 20 41 00 00 00 00 f7
```

```
0000013C16 = "GuitarProc"
0000010716 = "SpkSimType" = CAB
41 20 00 0016 = 10.0 = Boutique 2x12
```

Assignment CAB-names to Float values 0.0 to 16.0

Nr. (hex.)	CAB	Value field in Frames coming from THRII
00	British 4x12	00 00 00 00 =0.0
01	American 4x12	3F 80 00 00 =1.0
02	Brown 4x12	40 00 00 00 =2.0
03	Vintage 4x12	40 40 00 00 =3.0
04	Fuel 4x12	40 80 00 00 =4.0
05	Juicy 4x12	40 A0 00 00 =5.0
06	Mods 4x12	40 C0 00 00 =6.0
07	American 2x12	40 E0 00 00 =7.0
08	British 2x12	41 00 00 00 =8.0
09	British Blues	41 10 00 00 =9.0
0a	Boutique 2x12	41 20 00 00 =10.0
0b	Yamaha 2x12	41 30 00 00 =11.0
0c	California 1x12	41 40 00 00 =12.0
0d	American 1x12	41 50 00 00 =13.0
0e	American 4x10	41 60 00 00 =14.0
0f	Boutique 1x12	41 70 00 00 =15.0
10	Bypass	41 80 00 00 =16.0

These values are simply the IEEE-754 Floating Point representation of the values 0.0 to 16.0

Global parameters

If a *global* parameter is changed on the THRII, it sends out a parameter change message with opcode 04000000 as well.
In this case, the UKey field has got a value of $FFFFFFFF_{16}$, a kind of "pseudo unit key".

Example:

```
f0 00 01 0c 24 02 4d 00 6a 00 01 07 00 04 00 00 00 10 00 00 3c 00|7f 7f 7f 7f
4b 01 01 00 00 04 00 00 00 17 60 16 16 3e|00 00 00 00 f7
```

```
FFFFFFFF16 Global parameter
0000014B16 Parameter "AudioVolume"
3E96969716 = 0.294118 => 29.4
```

Some of the possible global parameters are "AudioVolume", "GuitarVolume", "GuitInputGain", "TunerEnable".

Reacting to manually changed User Setting on THR30II

If the user presses one of the 5 User Memory push buttons on the THRII, a message from THRII to PC is sent out:

```
THRII:                                                  Opcode      Lenght          Enum (?)      Number
f0 00 01 0c 24 02 4d 00 49 00 01 07 00 02 00 00 00 10 00 00 00 00|02 00 00 00 04 00 00
00 00 02 00 00 00 00 00 00 00 00 00|00 00 00 00 f7
      Enum(?)     Direction
```

Opcode 02 always has something to do with User Memory Settings and also occurs when patches are uploaded or downloaded. The local activation of a User Setting by pressing one of the push buttons is quite comparable with uploading a patch to the THRII. So, using the same opcode here makes sense.

The Length field tells us that there are $10_{16} = 16_{10}$ bytes to follow as the payload of this frame.

00000004_{16} Number of the selected User Setting (0-based => here U5)
(Value $FFFFFFFF_{16}$ would stand for "actual settings" but cannot occur, if this message results from pressing the buttons!)

The last parameter shows,
if the setting is uploaded to the active settings (code 00000000_{16}) or
downloaded from the active settings (code 00000001_{16}).
Invoking a setting from the User Memory is a download to the active settings, so it's a 00000000_{16} here.

With this message the Remote is briefed about the invocation. So far, so good. But what does the Remote has to do in this case?

To my mind, the Remote PC should have all information it needs. Resulting from the syncing process, it should know all parameter values of the invoked setting. So it should simply copy all the parameter values from the internally stored U5-settings to the "shadow" of the actual settings, it always keeps in PC memory.

But THR Remote instead makes THRII send all the settings again after asking if settings were changed.

```
PC:  (Short System Question: "have user settings changed?")
f0 00 01 0c 24 02 4d 00 60 00 00 07 00 0f 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f7

THR:   (answer: No, user settings have not changed)
f0 00 01 0c 24 02 4d 00 4a 00 00 08 00 01 00 00 00 01 00 00 00 00|00|00 00 00 00 00 f7

PC:    Request actual user settings
f0 00 01 0c 24 02 4d 01 0c 00 00 0b 00 0c 00 00 00 04 00 00 3c 00|7f 7f 7f 7f|00 00 f7
```

An actual settings dump follows from THRII to PC.

Changing parameter values

The main goal for my pedalboard may be to send whole patches. But of course you can change single parameter values as well.
Perhaps you want to add an expression pedal? It would be no problem for the Teensy to check the resistance of the potentiometer and to use this to control parameters like Gain, Master, delay time or a global parameter like Guitar Volume. I already consider to do so, because I have an old expression pedal of a keyboard lying around.

In this case you will have to send individual parameter change commands.

For "normal" parameter like setting "Master" to value 49.2 you will need an A-command with opcode 0A for this purpose. Remember, that you will have to split up the command into a header and a body frame:

PC: (Header with opcode 0a and 10_{16} =16_{10} bytes following in the body)
f0 00 01 0c 22 02 4d 00 5a 00 00 07 00 0a 00 00 00 10 00 00 00 00 00 00 00 00 00 00 f7

PC: (Body frame with UKey, PKey, Type and Value field)
f0 00 01 0c 22 02 4d 00 5b 00 00 0f 00 0c 01 00 00 4c 00 00 03 00 04 00 00 00 16 67 40 7b 3e 00 00 00 00 00 f7

010C = "Amp" = Main Controls
004C = "Master"
$3EFBE796_{16}$ Float 0.492001 => 49.2

Quick Start Guide

Not everyone will be willing or able to use the complete protocol and to properly react to incoming messages from THRII to PC and send complete patches to the THRII.

For simple remoting some settings like the selected AMP, CAB or one of the 5 stored presets a smaller approach could do. For example you can use programs like

"**Pocket Midi**" (Morson Japan) , https://www.morson.jp/pocketmidi-webpage/
Or
"**Send SX**" (Bome Germany), https://www.bome.com/products/sendsx
to switch some settings.

At least you have to get the THRII's USB-MIDI-Interface activated:

```
//UNIVERSAL_SYSEX Identity Request, non real time
F0 7E 7F 06 01 F7
//ask Firmware Version (00-variant)
f0 00 01 0c 24 02 4d 00 00 00 00 07 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 f7

//Header for sending magic key (4-Bytes frame follows!)
f0 00 01 0c 24 02 4d 00 01 00 00 07 00 04 00 00 00 04 00 00 00 00 00 00 00 00 00 f7
//1.43.0b body with magic key
f0 00 01 0c 24 02 4d 00 02 00 00 03 28 72 4d 54 5d 00 00 00 f7
```

After these 4 Messages your THRII's MIDI should be activated. You can proof this by operation a button on the device. If activated MIDI-IN data monitor shows incoming MIDI-SYSEX data every time you turn a button.

Now you can send some messages to change settings on your THRII.
For example changing **AMP** to "***Classic Lead***" is done by sending a header message followed by a body message:

```
f0 00 01 0c 24 02 4d 00 03 00 00 07 00 08 00 00 00 08 00 00 00 00 00 00 00 00 00 00 f7
f0 00 01 0c 24 02 4d 00 04 00 00 07 04 0c 01 00 00 19 00 00 00 00 00 00 00 00 00 00 f7
```

In the body message the 19 is the bitbucketed value for "***Classic Lead***", that is 99 (hex) before stealing its top bit and putting it to the bit bucket-byte of this 7-Byte-group. For here only this value has a set top bit, resulting in the bit bucket being 04 in this case.
If "Classic Special" had been your choice, you had sent this slightly modified body message:

```
f0 00 01 0c 24 02 4d 00 04 00 00 07 00 0c 01 00 00 78 00 00 00 00 00 00 00 00 00 00 f7
```

In this body message the 78 is the bitbucketed value for "***Classic Special***", that is 78(hex). Because in this case the top bit is not set the bit bucket is 00 in this case.

Besides this bit-bucketing changing settings is not that hard. But beware, that without drawing the symbol table form the THRII you are nailed to the actual firmware. In older versions the unit key 0x0000010C could differ and the values for "Classic Special" and "Classic Lead" could differ as well.
As shown in a table above the name for "Classic lead" in the symbol table would be "THR10X_Brown1" and the name of the unit key to change the AMP Model in the symbol table is "Amp"
To be sure to use the correct keys, drawing the symbol table from THRII is better than coding fixed keys in your SysEx messages.