Thimeo MicroMPX user manual

This document describes how to configure MicroMPX, and how to get the best results out of it.

Overview

MicroMPX or µMPX is a codec that transfers a full FM composite or MPX signal, meaning audio plus stereo pilot and RDS, over a low bitrate connection. It currently supports bitrates from 320 upto 800 kbit/s, and bitrates down to 176 kbit/s if you're using MicroMPX+ mode.

MicroMPX was developed specifically for use on FM, and even though the bitrates are low, it does not introduce typical lossy compression artifacts such as pre- and postringing or watery sounds. It also maintains peak control. If you use a composite clipper, the extra loudness that composite clipping generates also survives the MicroMPX codec. So for all relevant purposes, there's no real difference between connecting the direct composite output of a processor to the FM transmitter and connecting the MicroMPX decoder output to that same transmitter. (It *is* a lossy codec so the signal is not identical, which can become relevant when using a Single Frequency Network – more about that later.)

MicroMPX only needs one-way communication (from the encoder to the decoder, typically from the studio site to the transmitter site). This means that it can be sent over connections such as satellite links. It has several redundancy mechanisms to handle network or IP link problems: it can add recovery data to recover lost packets and send the same data over multiple connections so that as long as one of the connections works, the signal keeps playing. It is also possible to use multiple encoders that send their data to one decoder, to handle problems on the encoder end.

One encoder can feed any number of decoders (depending on the available bandwidth), and network multicasting or broadcasting is possible.

With MicroMPX, you can encode the full MPX signal in one location, and just spread it from there to all your transmitters, which will all get the same signal at the same time.

Warning: Sending MicroMPX over an unreliable connection such as the public internet may work perfectly fine, but it can also cause dropouts. If possible, use a reliable connection, or redundant connections.

Simple setup

(TODO: Image: 1 encoder, internet/link, 2 decoders with 2 transmitters. Basically Hans Rutten's image)

Redundant setup

(TODO: Image: 1 encoder, multiple links, 2 decoders with 2 transmitters.)

Redundant setup on both sides

(TODO: Image: 2 encoder in different locations, multiple links, 2 decoders with 2 transmitters.)

Single Frequency Networks: GPS synchronization
(TODO: Image: GPS receiver connected on both ends, show Simple setup with 2 decoders)

Simple setup

So, let's get started. In this document we're going to follow a “logical” order to set things up, which means that we first install the decoder (since that's easy, and without having a decoder it's impossible to see anything working), and then the encoder. We start with a local setup (local network), and after that we'll go into what's needed to make it work over an internet connection, through a router, and how to harden the signal against network issues. Finally we'll describe how to connect the decoder to an FM transmitter, and how to set up the levels correctly.

MicroMPX Decoder software installation
Install and run the decoder software on the system that you want to use for decoding.
System requirement: The system MUST have a 192 kHz capable output sound card.

- Microsoft Windows
  Run the Windows MicroMPX installer.
  From the Start menu, select MicroMPX Decoder.
  Click on the MicroMPX tray icon to open the web interface. It will usually open http://127.0.0.1:8080/

- ARM based device (like Raspberry Pi) running Linux
  This requires some knowledge of Linux.
  You only need a Linux version with a console, no desktop environment is required.

  Copy the MicroMPX_Decoder_ARM(XX) file to your ARM device.
  Make the file executable. You can do this by typing
  `chmod 755 filename`
  where filename is the name of the file that you just put on your device.
  Type
  `ifconfig`
  to find out the IP address of the Pi (you'll need this in a moment).
  Start the application by typing its name with `./` before it.
  `./filename`
  Next, open the web interface from a different device by entering the IP address of the Raspberry Pi. For example http://192.168.1.15:8080/ .

- Raspberry Pi 2/3/4 with HifiBerry sound card with our image
  We have a ready-to-go Raspberry Pi + HifiBerry image available on our website.
  You can download the image and write it to an SD card with Win32DiskImager, balenaEtcher or similar software. You need an SD card of **more than** 4 GB.
  Put the SD card in the Pi and boot. After booting (twice), the MicroMPX Decoder will automatically start.
  On a different system, open the IP address of the Pi, for example http://192.168.1.15:8080/ .

  If you need to login on the system, the username is `mpx`, the password is `micro`.

After logging in, you can run

```
ifconfig
```

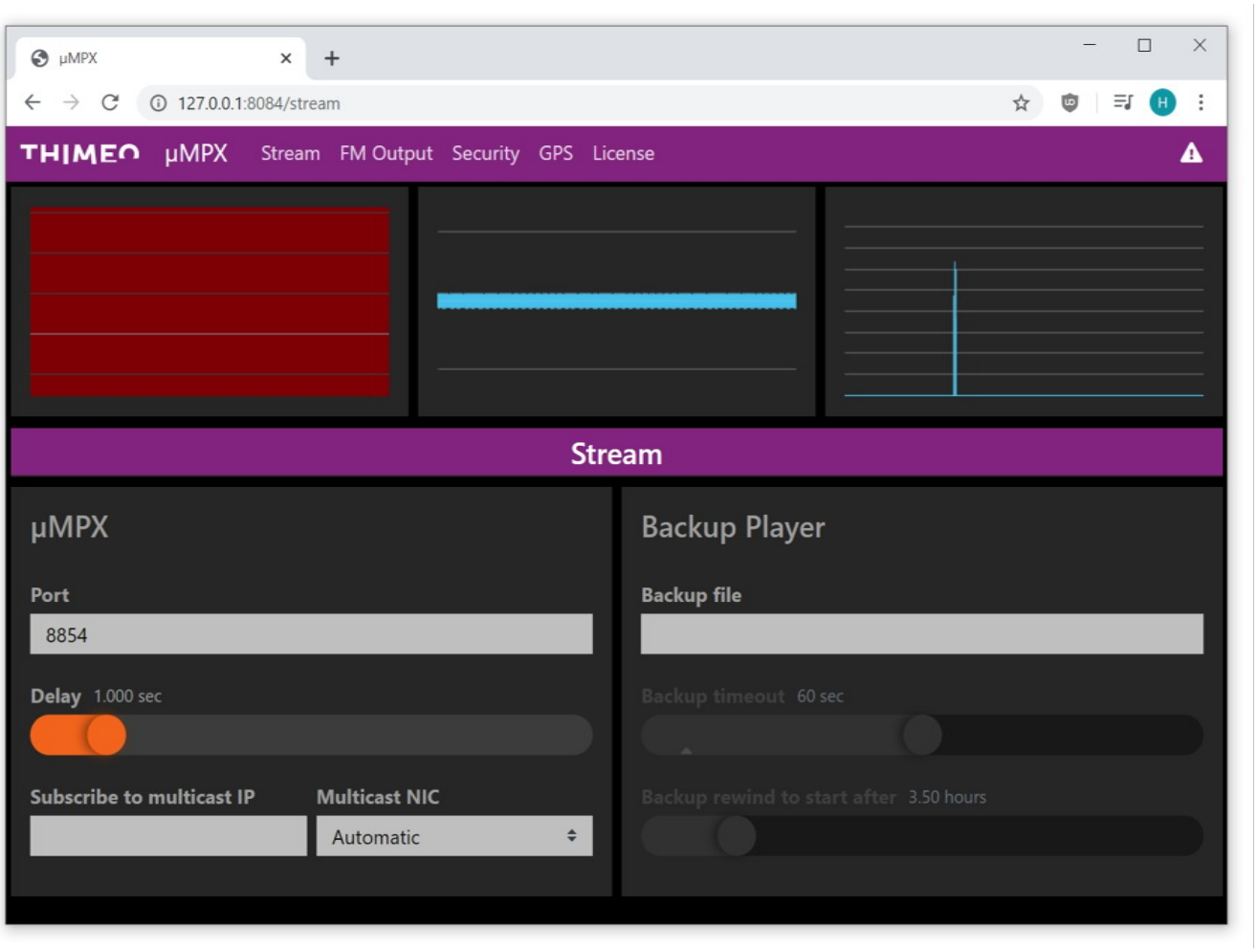
to find its IP address, and you can run

```
./update.sh
```

to download and install the latest MicroMPX version, or to switch between encoder and decoder. You need to reboot after doing this.

Please note: HifiBerry is a brand name. You cannot just put in a different sound card and expect our image to work – if you use a different sound card, you need to configure it yourself. In that case, use the standard Raspbian distribution instead of our SD card image.

At this point you should see the MicroMPX Decoder web interface. From now on the instructions are identical regardless what type of device you're running it on.



Before we start configuring the decoder, we'll first set up the encoder.

MicroMPX Encoder software installation

This step is only needed if you want to run our MicroMPX encoder. You can also use the MicroMPX

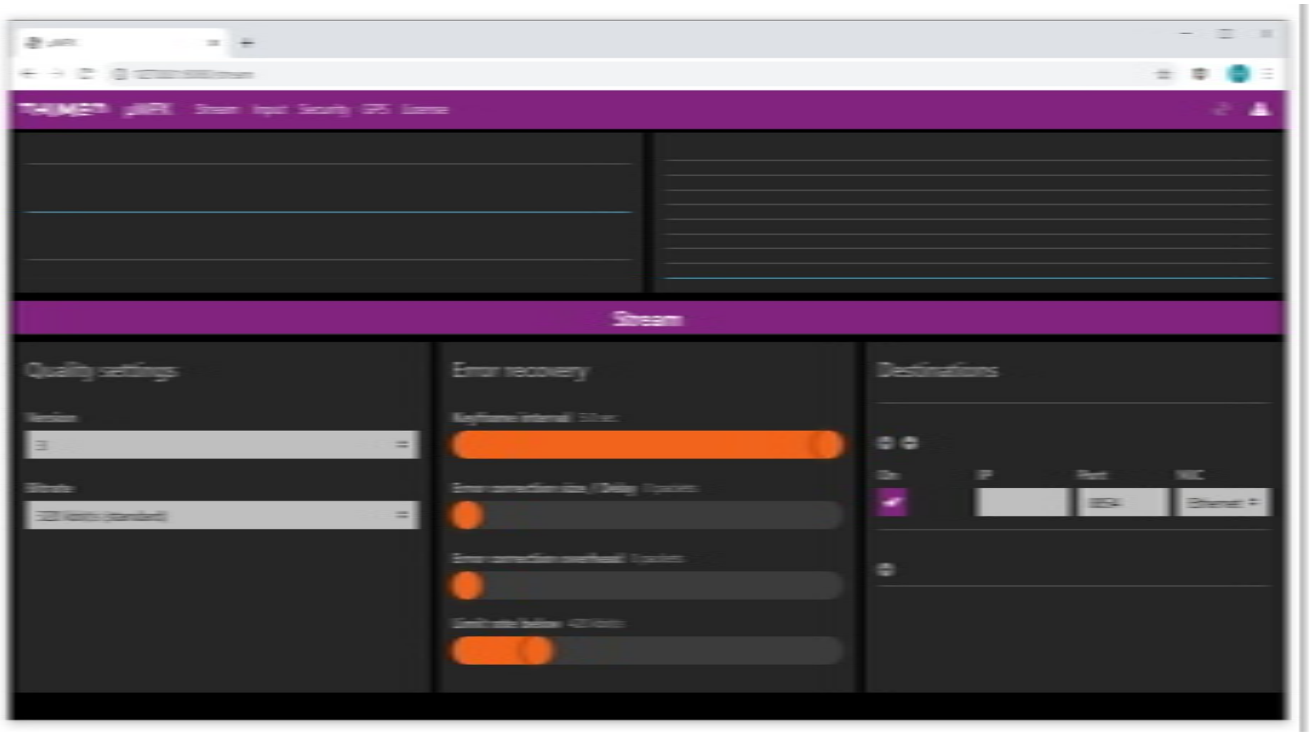
encoder in another device or in an audio processor that has a MicroMPX output, such as Stereo Tool or Omnia.9.

Install and run the encoder software on the system that you want to use for encoding.
System requirement: The system MUST have a 192 kHz capable input sound card.

The steps are basically the same as for the decoder. Except when you're using our Raspberry Pi image, in that case you need to login and run

```
./update.sh
```

as described above and select the encoder, then reboot.
Our image is setup for a HifiBerry DAC. If you're going to use the encoder, you probably have a DAC+ADC, or maybe some digital device. In order to be able to use it, you need to edit the file /boot/config.txt and put the device that you have in there. See https://www.hifiberry.com/docs/software/configuring-linux-3-18-x/ for a list of all available HifiBerry devices. For a DAC+ADC, replace “hifiberry-dacplus” by “hifiberry-dacplusadc” or “hifiberry-dacplusadcpro”, depending on which exact card you have. After changing this, you also need to reboot.



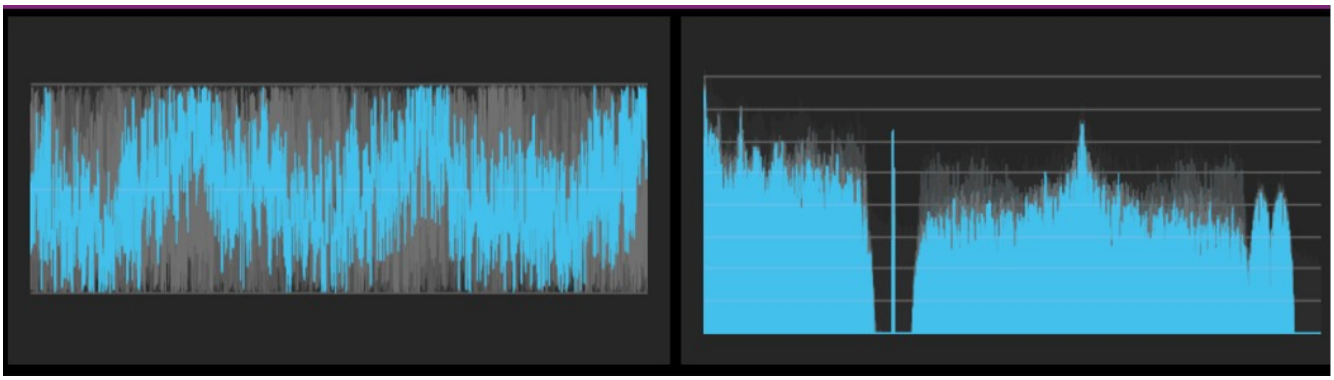
MicroMPX Encoder configuration: Getting it to work
Next, we need to tell the encoder two things: Where to get its MPX audio from, and where to send it to.

In the Input tab, you can select an input sound card.

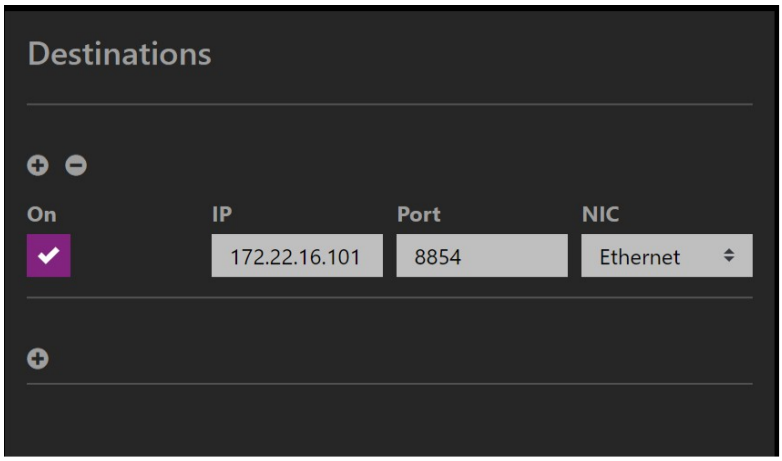
Important: For MicroMPX to work correctly, the input level has to be at exactly 0 dB. Please adjust the gain slider if needed such that the thin lines in the scope display just *reach the peaks.*

Important: If possible, use a digital sound card. Analog sound cards usually have a highpass filter in their inputs, which causes the peak level to become non-constant. If needed, the "Input tilt" correction can be used to compensate for this. This is never needed for digital inputs, but it usually is needed for analog inputs.

These two remarks only apply when you use a MicroMPX encoder that's not built into an audio processor.



Once done the display should look like this:



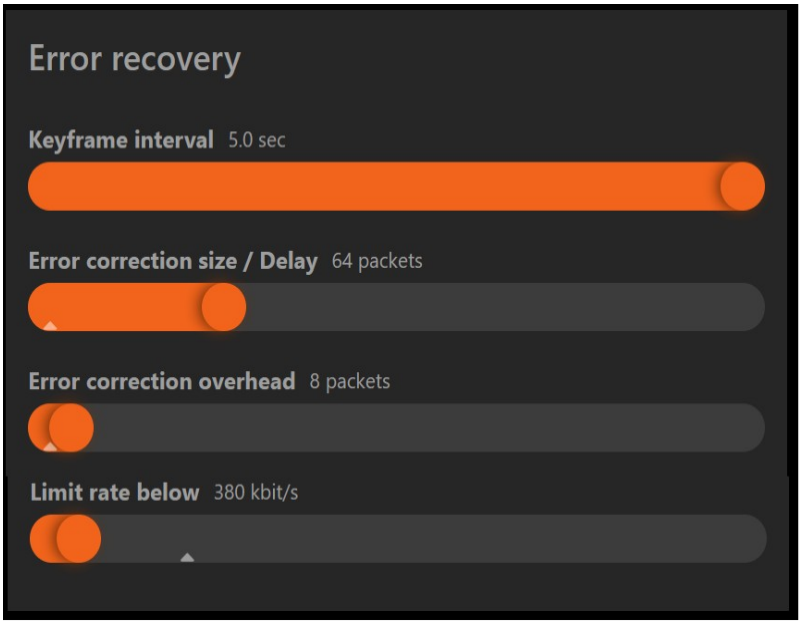
Next, type in the IP address of the decoder device in the Stream window:

The default port for MicroMPX is 8854. If you changed it in the decoder, make sure to use the correct port in the encoder. You should see the meters move in the decoder now as well, showing the same waveform and spectrum data that the encoder is sending. You may need to select an output sound card in the decoder, although by default it should have picked the first one that it found that can play back 192 kHz audio.

MicroMPX Encoder configuration: Forward error correction

MicroMPX streams over UDP, which is susceptible to packet loss on busy or unstable networks. It contains multiple mechanisms to ensure stable streaming even when packet loss occurs.

The easiest one to use is Forward Error Correction. Here's what a typical setup looks like:



What these settings say is: “Every 64 packets, send 8 extra packets as overhead.”. For each original 64 packets, 72 actual packets are sent. As long as at least 64 of the resulting 72 packets arrive, the decoder can reconstruct all the missing packets.

The resulting bitrate is slightly higher than the configured bitrate. For example, when the bitrate is set to 320 kbit/s, and 64/8 is used as recovery settings, the resulting bitrate is 320 + 320 * 8/64 = 360 kbit/s.

Network dropouts typically occur in bursts, so typically you'll see a number of subsequent packets getting dropped. This means that in the end, it doesn't really make a lot of difference for how resilient the signal is against packet drops whether you set it to 64/8 or for example 32/8 or 128/8. You might think that 128/16 is similar to 64/8, but it's actually (almost) twice as powerful, at the same bitrate (320 + 320 * 16/128 is also 360).

You might wonder why there are two settings (64/8) and not just one (8). That's because the first value determines how much time there is between blocks of recovery packets. MicroMPX sends about 94 packets per second, not counting the recovery packets, so when you use 64/8 the latency of the decoder must be at least about 0.7 seconds. For 128/16 it needs to be 1.4 seconds. You should set the value a bit higher than this to account for small differences in speeds and the time it may take to send FEC packets; using 1 and 2 seconds in these example cases is definitely safe.

As an example, let’s say that according to the error logs, the biggest drop that has been reported was a drop of 5 packets. In that case, you’ll want to make sure that all the drops of up to and including 5 packets will be recovered, so the “Error correction overhead” must be set to at least 5 packets. You might want to set it a bit higher to also recover slightly longer dropouts. If you want to keep your delay at at most 1 second, you can set the “Error correction size” to around 64 packets. If you don’t mind having more latency, you can set that value higher without really affecting recovery, as described above.

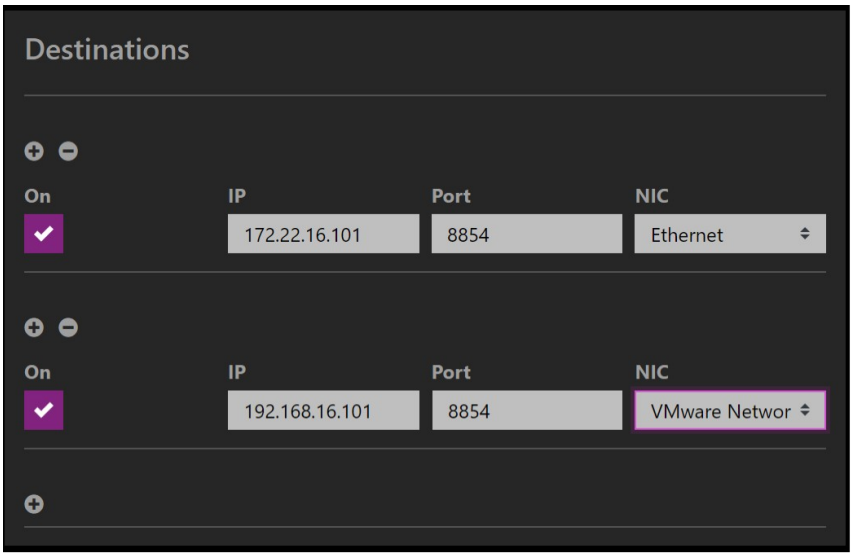
The Keyframe interval determines how often a keyframe packet is sent. If the connection is really lost, or if the decoder switches to a different stream or is initially started, playback can only (re)start on

keyframes. *If* a packet would be lost, playback will halt until the next keyframe is received. Setting the time between keyframes lower will therefore result in shorter dropouts if a dropout occurs. Setting the keyframe interval lower will leave slightly fewer bits for data, but it doesn't really have a big impact, so you can safely set it to 0.5 seconds for example.

Finally, Limit rate below controls how fast packets can be sent through the network. *You should normally just leave this at the default setting.* All recovery packets in a block are generated at once. If the number of recovery packets is high, this can lead to many packets being sent to the network at high speed. This could in itself cause packet loss. So, without rate limiting, adding a lot of recovery packets would actually make the connection less reliable. Because of this, it's usually best to leave this setting slightly higher than the theoretical value calculated above (the bitrate multiplied by the error correction factor), and always lower than the maximum network capacity available.

MicroMPX Encoder configuration: Redundant links

In the encoder under Stream, you can add more destinations. For redundancy, if you have multiple connections between encoder and decoder (for example two different ISP's, a satellite connection and a link, or any combination), you can send copies of the same stream via different links. This way, as long as a packet arrives via any of the links, it will be received and decoded.



In most situations, the network layer of the operating system can determine based on the target IP address which NIC (network interface, basically which network connector) to use. If you want to explicitly send the data over multiple networks it's not unlikely that an IP address is reachable via multiple connections. To force the encoder to use two different NIC's, you can explicitly specify which NIC to use.

There is also another way to add redundancy. A MicroMPX decoder will “lock onto” a stream that it's decoding. So even if another stream is sent to the same port of the same device, it will keep playing the stream that it's playing. But, if that stream drops out, it will start decoding again at the first keyframe that it receives. That can be a keyframe of the same stream, but also of another stream that's sent to the same decoder port. This makes it possible to have 2 encoders, for example in different locations, that encode the same signal. If one of the two drops out, the decoder will automatically switch to the other

one.

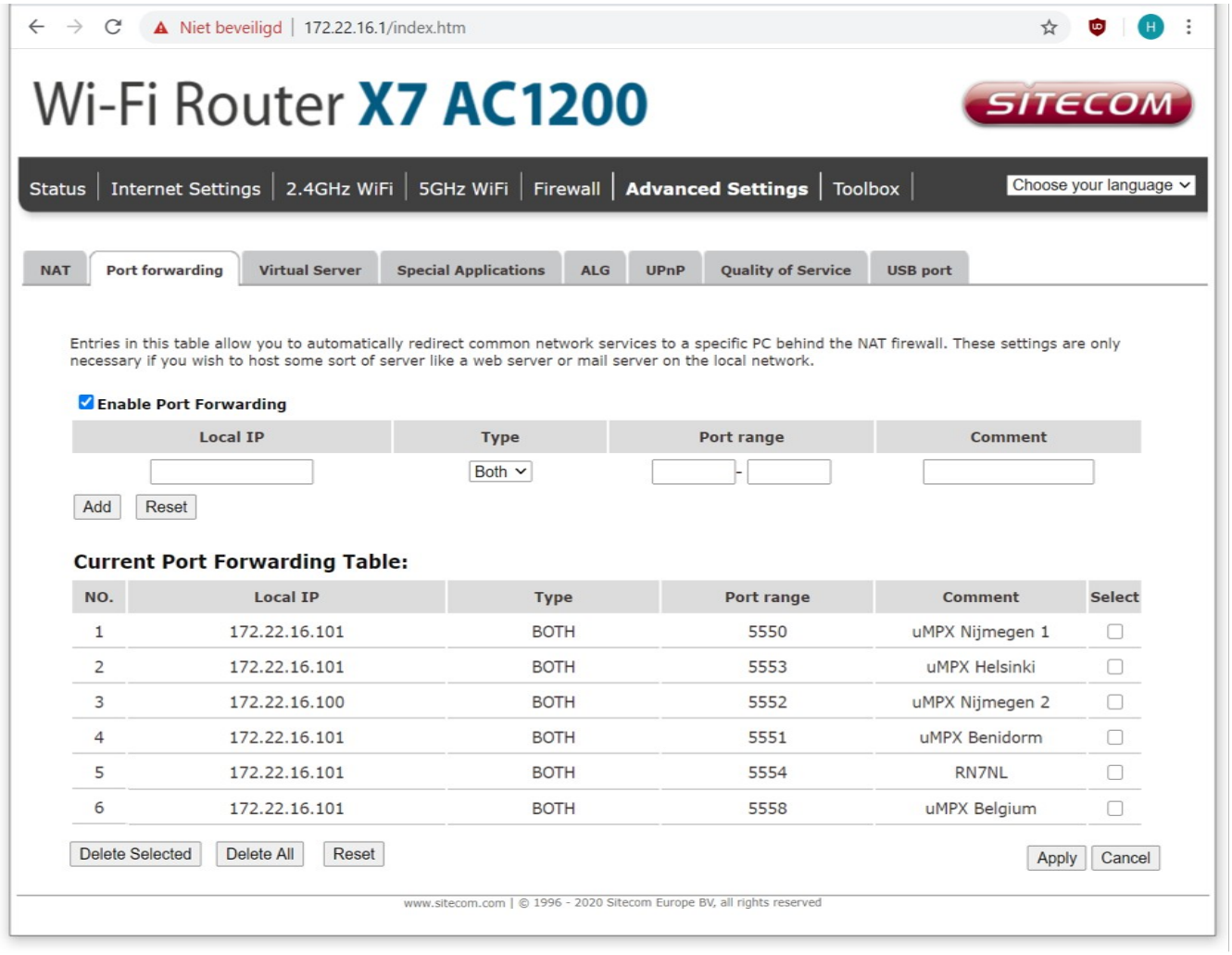
MicroMPX Encoder configuration: Multiple decoders
You can use the same mechanism to send data to multiple decoders. At this moment, upto 100 IP addresses can be used.

As long as you keep the Delay setting in the decoders the same, and there are no large differences in delays in some of the paths (which might be the case for example for a satellite link), all decoders will play the audio at the same moment in time within a few milliseconds, which is good enough for seemless RDS AF frequency switching. If a link does have a delay, the Delay setting for that decoder can be adjusted to compensate for it. If you need the signals to be even more accurate for Single Frequency Networks, see below.

Port forwarding and networks
So at this point you're ready to take the decoder to your transmitter site and connect it to your transmitter. If that's on the same local network as the studio, you can skip this step. But if the data goes over the internet, or another router, you probably need to configure port forwarding: You need to tell the router that UDP packets that are targeted at the decoder must be allowed through. And where they need to go.

To do this, you need to login to your router, and setup port forwarding. Typical router IP addresses are 172.22.16.1, 192.168.1.1, 10.10.10.1. You can usually find your password to login on a sticker on the router. Then, find a section called “Port forwarding”, “Virtual server” or something similar. The latter often allows you to route data to another port number internally than externally.

This image shows a setup that sends incoming signals from ports 5550-5558 to different devices in the local network:



If you want to find out what the IP address of the decoder is (which you need to fill in in the encoder), a site like https://whatismyip.com/ can tell you that – just open it from the decoder and it will show you its public IP address.

MicroMPX Decoder configuration: Connecting to an FM transmitter and calibrating

How to connect the decoder to the transmitter depends on several things. If the transmitter has a digital MPX input (MPX over AES/EBU), that is the preferred way of connecting it. And most of the steps below can be skipped in that case.

For analog MPX inputs, the transmitter might have an XLR MPX input, or a BNC MPX input. The sound card may also have a balanced (XLR or other) output or an unbalanced (usually RCA or minijack) output. If both sides support balanced I/O, that's the preferred way of connecting it. Otherwise there are lots of convertors available, for example from XLR to and from RCA, from RCA to BNC etc. You only have to connect one channel, since it's an MPX signal (basically a data signal).

After connecting it, you need to adjust the level such that the output of the transmitter complies to your local laws (75 kHz modulation – or sometimes a bit more, mainly). MicroMPX has a test tone generator, both for sine waves that can be used to setup the level and if needed boost the high frequencies a bit if there is some high frequency rolloff, and a square wave generator that can be used to compensate for tilt in low frequencies, typically caused by a highpass filter (DC removing filter) in

many sound cards, and sometimes even in older transmitters.

Step 1: Select an approx. 1000 Hz tone, and adjust the level (either the MicroMPX output level or the transmitter input gain) to match the maximum allowed modulation.

Step 2: Switch to a 30 Hz or so square wave, and adjust the RC slider until the modulation is as close as possible to that same maximum level. Go down to 15 or 10 Hz to adjust it more accurately. Normally, you should be within at most 1-2 kHz of the allowed maximum once this is done, more than that indicates some issue.

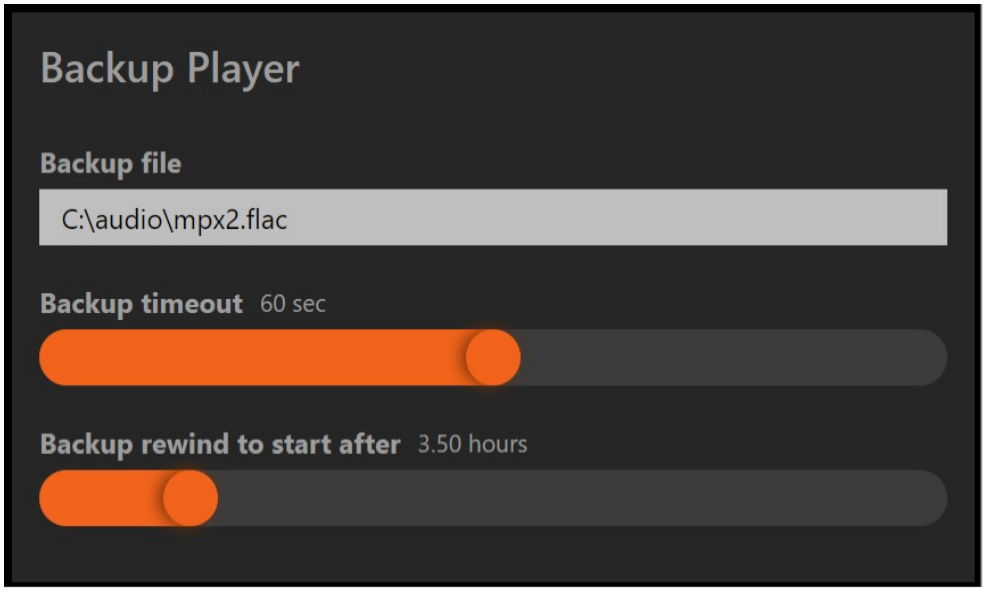
Step 3: Check how much the level drops at for example 50 or 60 kHz. And adjust the Highs RC slider – CAREFULLY because it's very easy to overdo it. Having too little highs causes no real problems except some loss in stereo separation and RDS level, but having too much can cause overshoots. *Important: when using the tilt correction or Highs RC correction in the MicroMPX decoder, its output level MUST be set below 0dB, or digital clipping could occur. Check the waveform display to make sure the waveform never exceeds the lines on top and bottom.*

MicroMPX Decoder configuration: Backup audio

So you have everything running, but what if the connection drops out? Then your listeners will be enjoying silence. Fortunately, there's a solution: You can prepare a file with emergency audio, which will be played in this case.

The MicroMPX decoder expects a 192 kHz mono file that contains MPX data, normalized to 0 dB, in WAV or FLAC format. The file is transmitted as-is, so it needs to be pre-emphasized and it must contain stereo and RDS signals.

MPX files can be generated by recording the output of an FM processor (or even the received signal from a tuner that doesn't demodulate the MPX signal). Or you can use a program like Thimeo WatchCat to generate an MPX file from an audio file.



The backup player kicks in after Backup timeout time of no usable MicroMPX data coming in. If a few packets arrive and then the signal disappears again, it will continue the backup file playback where it left off, unless the backup player hasn't been used in Backup rewind to start after time. In that case it will start at the beginning. If you always want it to start at the beginning (for example if you have a loop of jingles as backup file), you can set that time to 0.

MicroMPX version and bitrate
We started MicroMPX development in 2015, and have kept improving on it. Due to this there are now 4 different versions of the MicroMPX bitstream.

All encoder versions of MicroMPX can generate streams in older versions than their “preferred” version. And all decoders can decoder older streams. But not newer ones. So if you update the encoder first and want to keep the existing decoders working, you need to configure the encoder to generate an older version bitstream. You can switch to the new bitstream when all decoders have been updated. You can also update the decoders first.

We advise to *always* use the latest version as it contains fixes for issues found in previous versions. We haven't broken compatibility unless there was a good reason to do so.

Version 1 supports bitrates from 320 upto 400 kbit/s.
Version 2 and 3 support bitrates from 320 upto 576 kbit/s.
Version 4 supports bitrates from 320 upto 800 kbit/s, and down to 176 kbit/s in MicroMPX+ mode.

Higher bitrates will generate an output that's closer to the original, but unless you compare the actual waveforms or spectra it shouldn't really be possible to tell any difference between them.

We have selected 320 kbit/s as the minimum bitrate based on multiple days of extensive listening tests with a large group of volunteers, which showed that they were unable to identify if a recording had been encoded or not in an A/B/X test – the results were within the error margin. See the section about MicroMPX+ below for a discussion of tradeoffs.

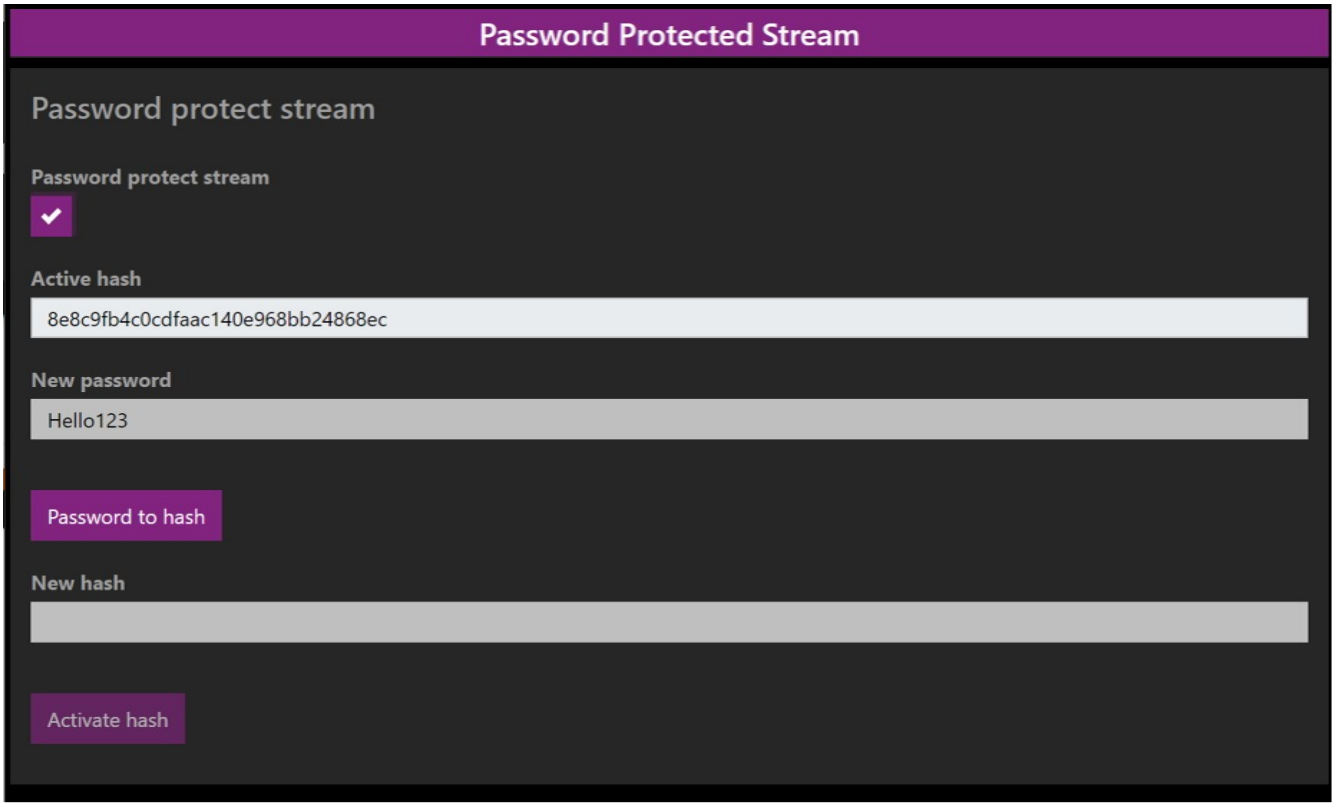
Security / Password protection
If you're streaming over a public internet connection and other people could potentially send data to the same IP:PORT combination, they could overrule your stream.

You can use a hashcode to protect against that. Only streams coming from an encoder which uses the same hashcode will decode as valid audio. A hashcode can be typed in directly, or generated from a “password”. Note that if someone has access to the web interface, they can copy the hashcode and override your stream (but they could do that anyway if they have access).

The hashcode must be identical in the encoder and all the decoders, any mismatch will cause the decoder to go silent, and you'll get lots of error messages because the packages that arrive will be decoded incorrectly and contain garbage. So if you turn this on, it must be enabled in both the encoder and all the decoders.

You can either generate the hash from a password (which can be easy to remember, so you can type it in on all units and hit “Password to hash”), or you can copy the hash itself, which is probably more difficult to remember, but it also means that if you forget the password and you have a network of 200

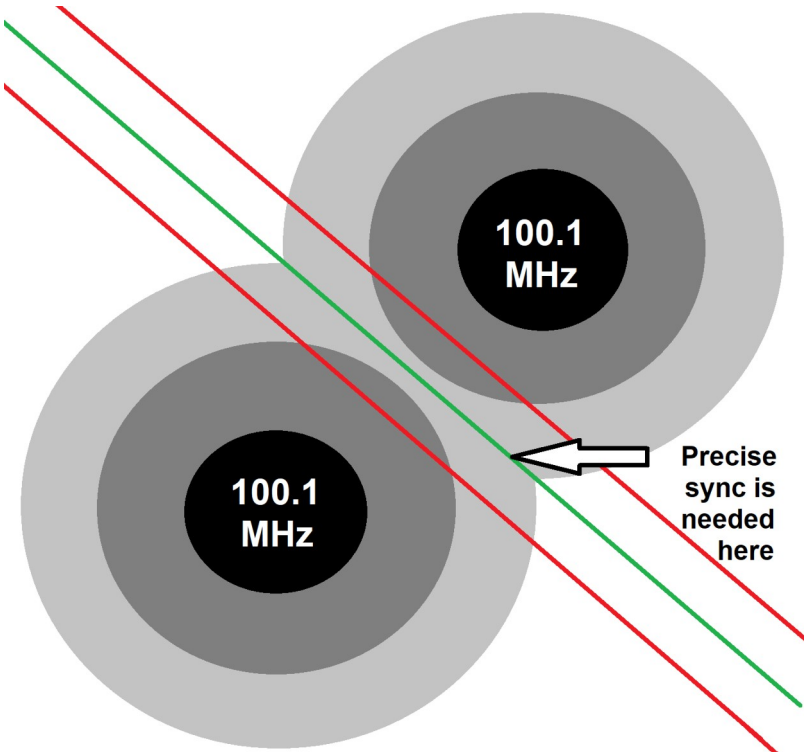
decoders and you want to add one, you don't need to set a new password on the other 200 units.



GPS synchronization

Due to COVID-19, several of the sites that were going to test our SFN synchronization have not yet been able to perform their tests. This means that we don't have as much real-world experience as we would like. All the tests that have been done showed that it works, but we don't fully know yet what type of problems users could run into. For now, please purchase this feature only if you know what you're doing and are able to aide us in debugging any problems.

MicroMPX will normally keep your decoders in sync within a few milliseconds, which is good enough for seemless RDS AF switching. But if you have a Single Frequency Network (multiple transmitters at the same frequency with overlapping reception areas) and you want to control exactly where the signals add up instead of interfere with each other, you need far more accurate timing. Typically a precision of less than a microsecond is required (one millionth of a second).



To do this, the MicroMPX encoder needs to add timestamps to the audio, and the decoders all need to precisely synchronize when they play the audio using a GPS clock.

Note: This currently only works when you use the software MicroMPX encoder or Stereo Tool as encoder. It is not supported from 3rd party hardware.

To make this work, you will need several things, which you will get when you buy MicroMPX Encoder or Decoder with SFN support licenses from us. There are ways of "building your own hardware", but if it doesn't work it will be nearly impossible to figure out why.

We have so far only verified that this works on a PC. You need to have a sound card with ASIO support, which guarantees that the input-to-output signal delay is always exactly the same. Also, we strongly recommend to use identical hardware on all the SFN decoder sites, to avoid other delays that need to be compensated for.

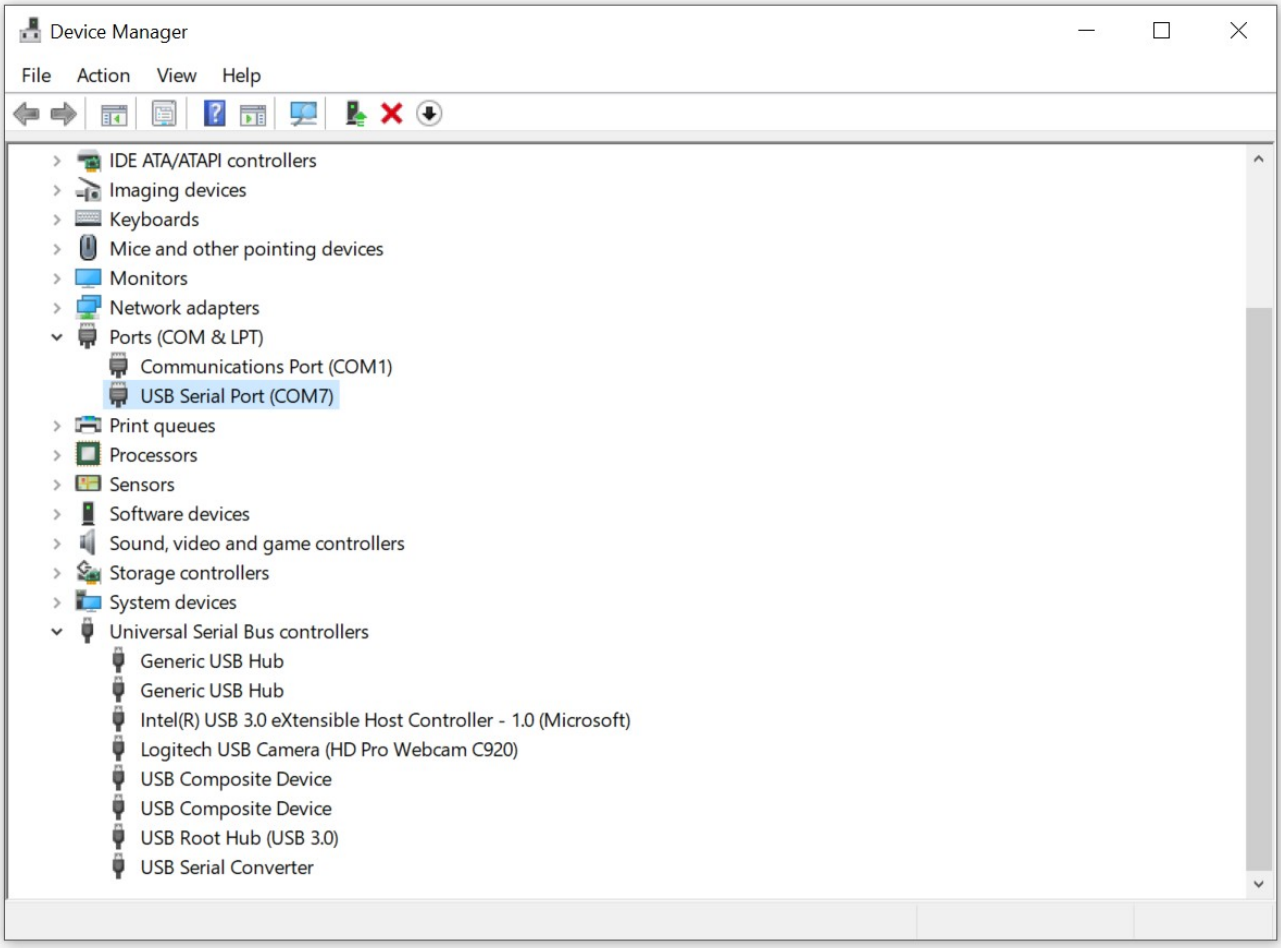
On the encoder end, all you need is a GPS receiver.

For each decoder, you need a GPS receiver with PPS (Pulse Per Second) support, plus a converter unit to connect the signal to to both the USB port and the sound card. If you order the license from us, we will provide both.

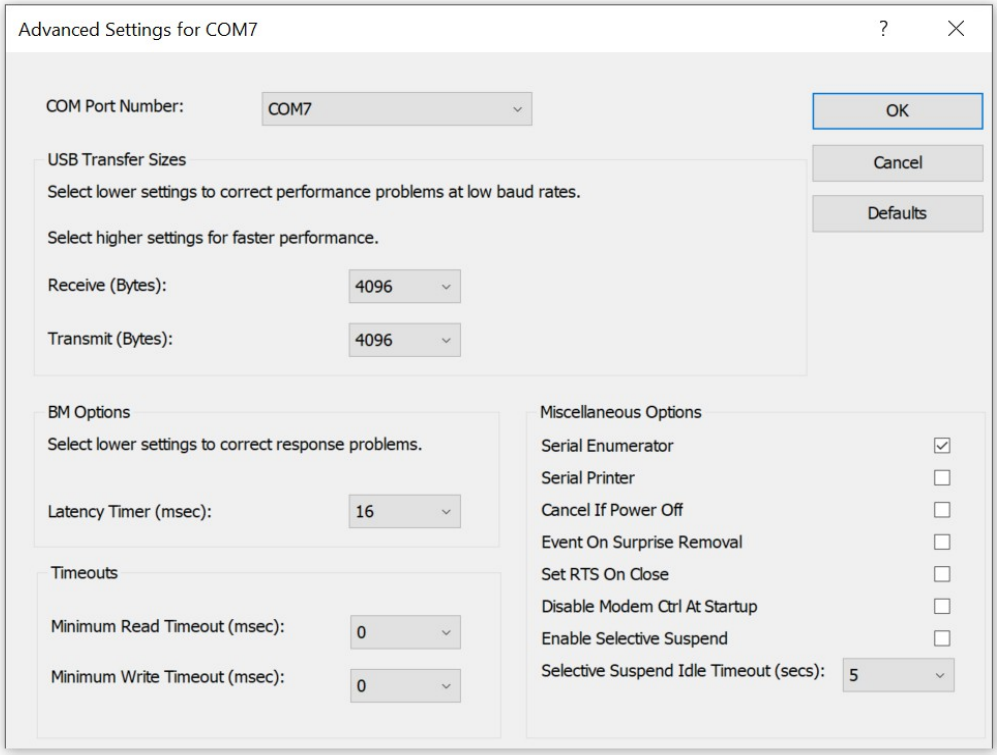
Encoder (our software stand alone encoder, or the encoder in Stereo Tool)
On the encoder system, connect the GPS receiver to a USB port. In principle, any GPS receiver that sends an NMEA signal to a COM port (which can be a USB port) *should* work (but it might not have been tested, and there could be issues). If you use our USB-based GPS receiver:

- Connect the USB cable to your computer.
- Place the GPS receiver somewhere where it can receive GPS signals, for example close to a window.

In Linux this is usually sufficient. If you're using Windows, you may need to go to Device Manager (click on the Windows Start icon, type “Device manager”, and click on the link). This should take you to the Windows Device Manager, which contains a section Ports (COM & LPT). If you open that section, you'll see that a USB Serial Port is added, and what its assigned port number is, in this case COM7:



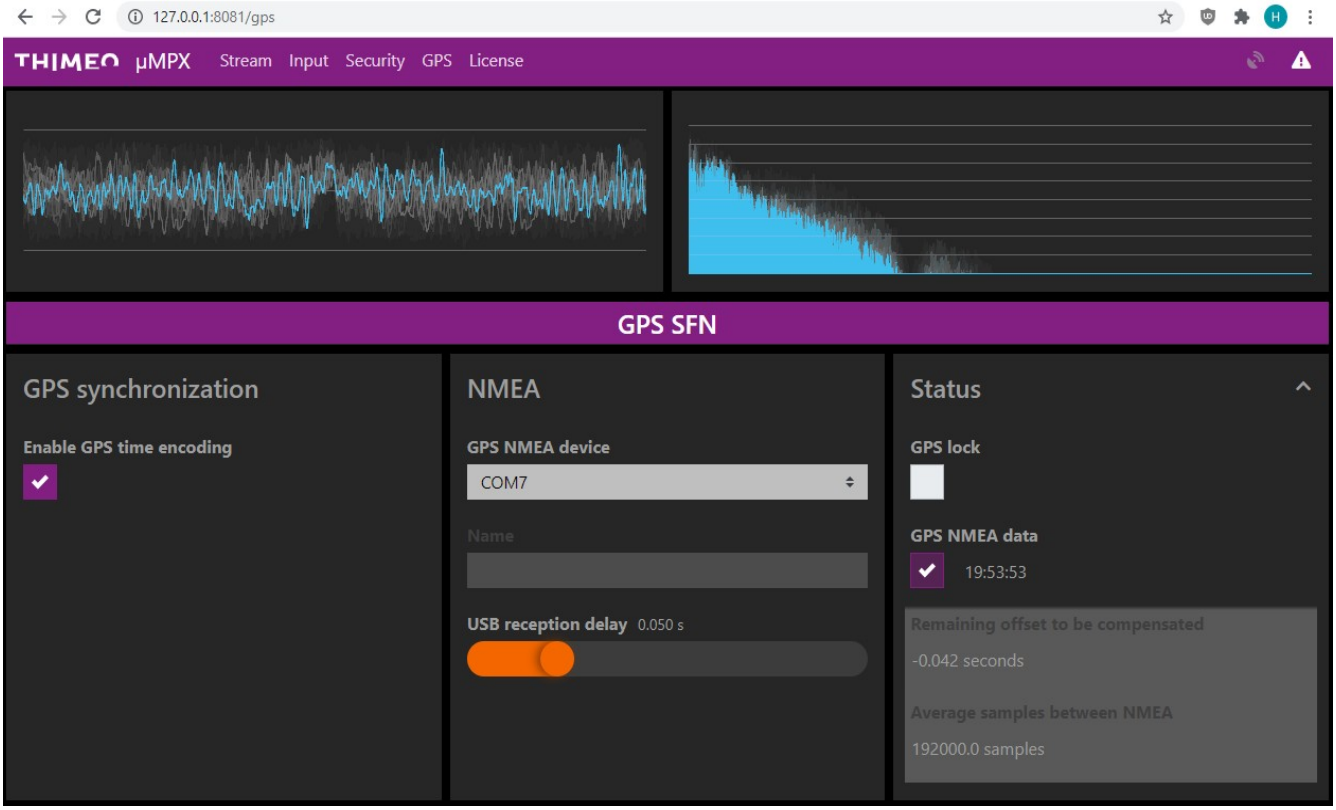
Double click on it, go to Port settings, and click on Advanced. That will open this screen:



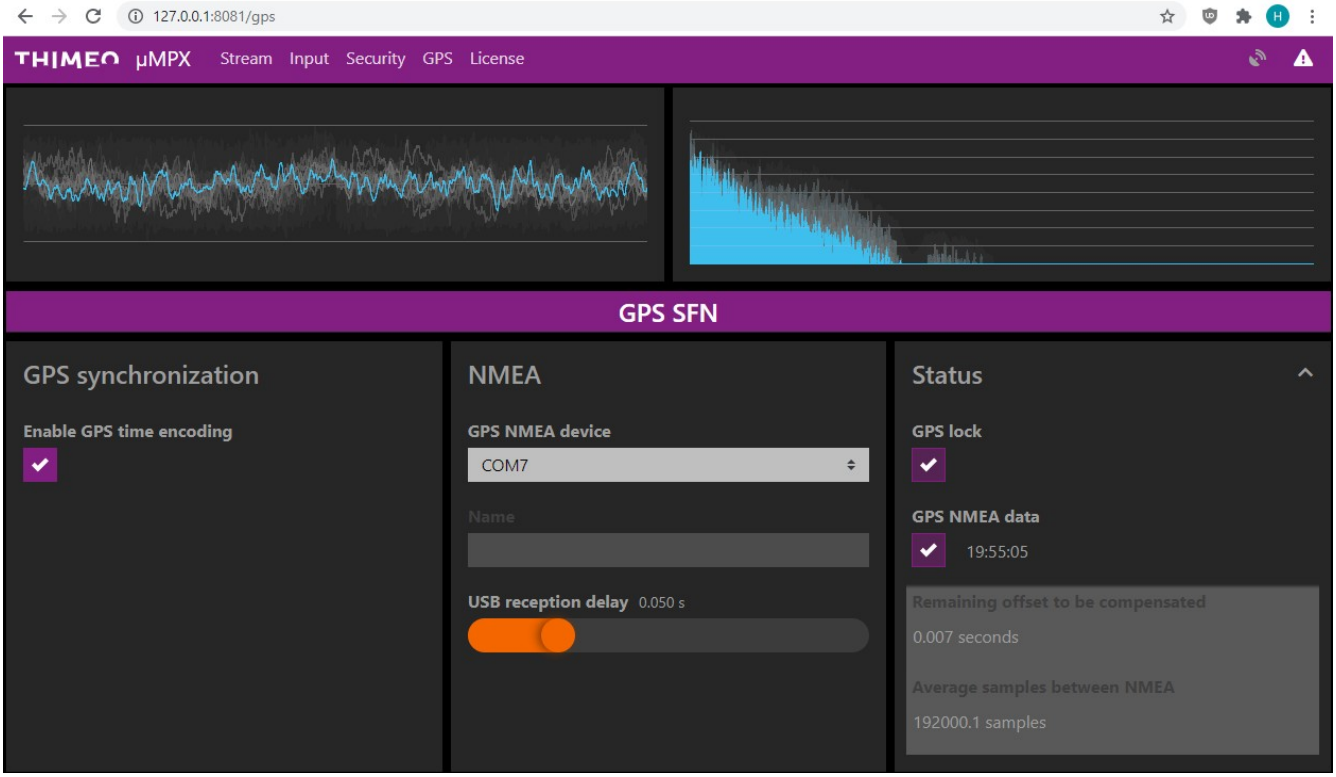
In some cases, once the GPS receiver starts receiving data, Windows might interpret it as mouse movements – if that happens you'll see the mouse jump over the screen. In that case, turn “Serial Enumerator” on or off.

In the MicroMPX Encoder web interface, go to GPS, enable GPS time encoding. When you do this, the GPS lock icon will appear in the right top of the screen, and it will look grayed out. Next, select the proper COM port – in this case, following the screenshots above, COM7. On Linux the name might be something like /dev/ttyUSB0, /dev/ttyUSB1 or – for some receivers – something like /dev/gps.

It may take a few minutes before the GPS receiver gets a lock, and it needs to be placed such that it can “see" enough satellites – so if you're inside a big metal building it might not work. Usually, placing it close to a window will suffice.



First, the GPS NMEA data will apear. After about 30 seconds, GPS lock will light up (both the name under Status and the GPS symbol in the top right corner of the screen):

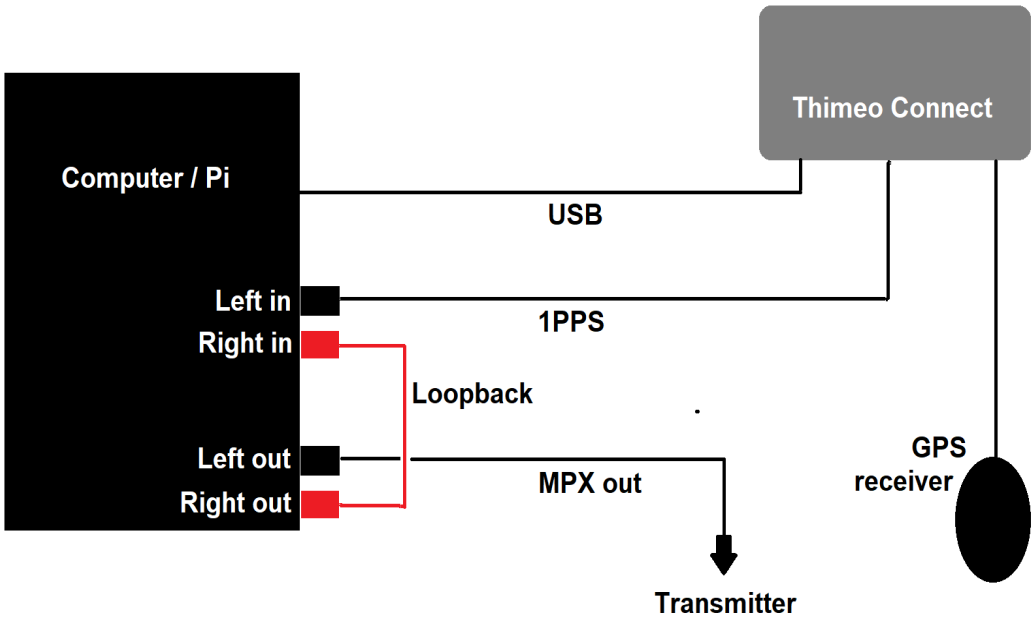


That's all for the encoder end, the MicroMPX bitstream now contains GPS timestamps.

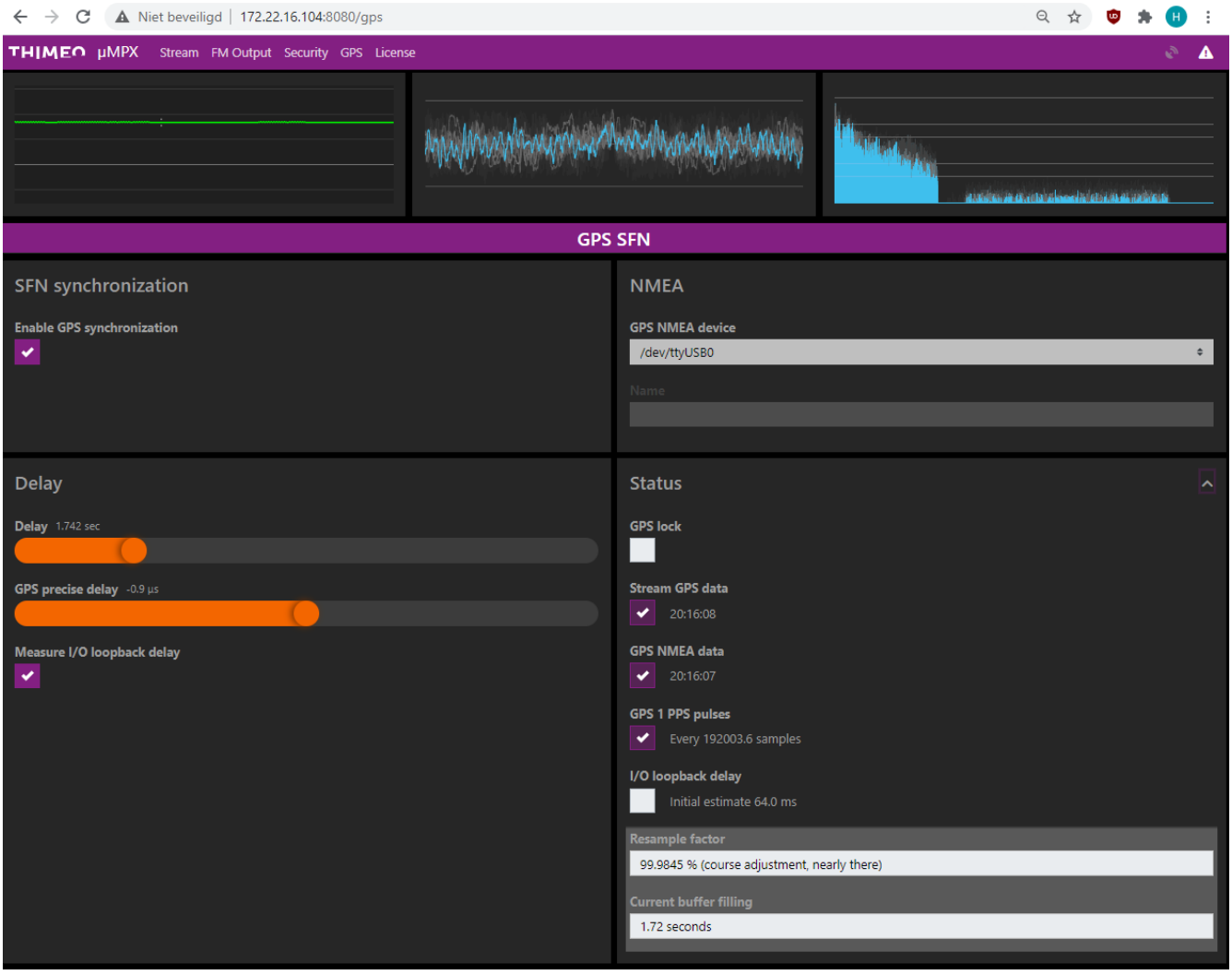
Decoder (our software decoder)
For the decoder, we sell a converter box with GPS receiver. Alternatively, you can use your own – the GPS receiver must be able to send NMEA data to a COM port (which can be done via USB) and a 1PPS pulse signal to the sound card.(Theoretically the sources can even be two separate receivers).

Connect the GPS receiver to the Thimeo Connect box, and connect a USB cable from the Thimeo Connect box to your computer. Then connect an audio cable from the Thimeo Connect box into the *left channel* sound card input that is coupled to the output that will be used for the MPX signal. If you're using Windows, use a sound card that supports ASIO, and select an ASIO sound card in the sound card dropdown list. Synchronization will not work without ASIO on Windows.

For some sound cards, the input-to-output delay is not perfectly the same on every boot. We have seen this for example with the HifiBerry DAC+ADC card on a Raspberry Pi, but also with a Steinberg UR22MK2 on a Mac. The difference can typically be upto a few microseconds. This is a problem, because after restarting you could get a different delay. To measure the exact input-to-output delay, we recommend connecting an extra cable from the right channel output to the right channel input and enabling “Measure I/O loopback delay” on the decoder. This also helps to prevent differences in delays between different sound cards.



Then, go to GPS in the decoder web interface and enable the SFN synchronization:

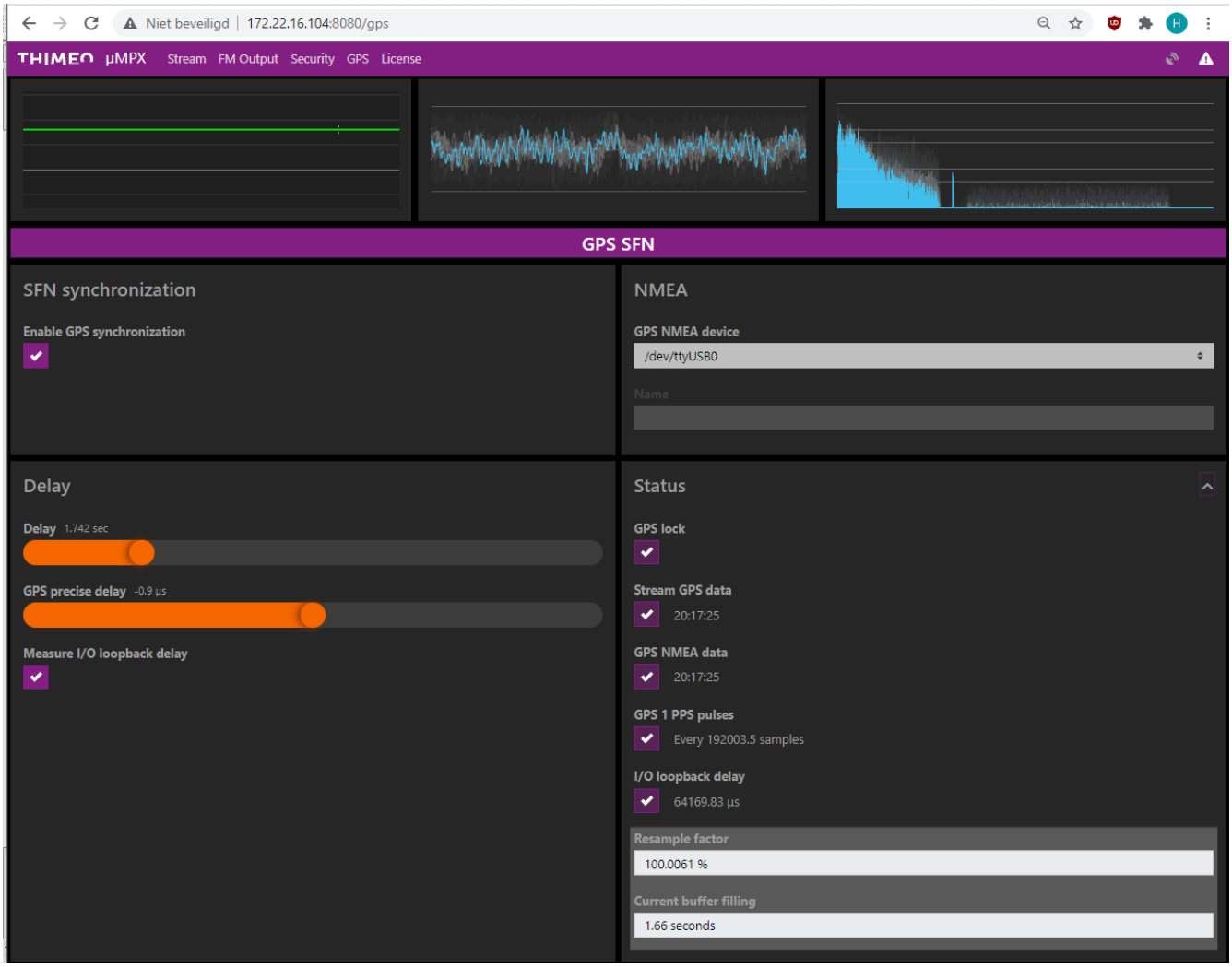


Initially, the GPS timestamps from the encoder, the timestamps of the GPS receiver on the decoder and the time (in samples) between 1PPS pulses will be displayed. If enabled, the I/O loopback delay will show an estimated delay time in milliseconds.

If the 1PPS pulses and loopback delay values don't appear after a few seconds or if they jump wildly, you may have swapped the left and right channels, so try swapping them. The right output channel (the one that contains the audio used for loopback delay measurements) sounds like a constant beep, so if you hear that where you expect your MPX signal, the output channels are probably swapped.

After about 30 seconds, the I/O loopback delay will light up and show an exact time (the measurment is accurate with a precision of 0.01 microseconds).

Windows only: If the selected sound card is not an ASIO sound card, at this point the audio will stop playing and you'll see a runaway in the stream display. This will be resolved as soon as you select a working ASIO sound card.



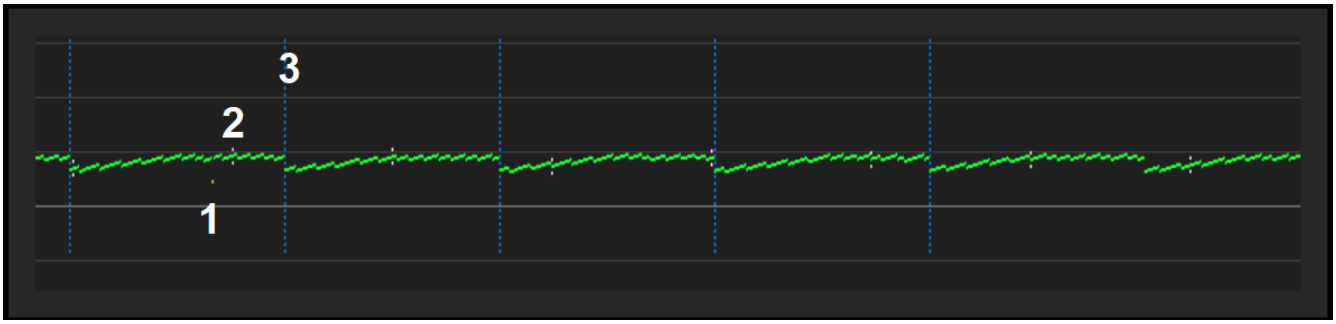
As soon as the GPS signal is recognized and synchronized to the signal from the encoder (so make sure

that the encoder is sending GPS timestamps), the GPS lock icon will light up as it did in the encoder. As soon as that's the case, usually within 2 minutes, if you have multiple decoders that are configured identically and show a GPS lock, the audio is synchronized. The offset in the delays of the decoders should now be constant within about 0.5 microseconds.

To fine-tune where the signals interfere and where they boost each other, you can use the “GPS precise delay” slider. Make sure though that the main “Delay” slider is set to the same setting for all decoders.

Understanding the MicroMPX Decoder stream info display

The MicroMPX Decoder web interface has a display that shows a lot of information about incoming stream packets.



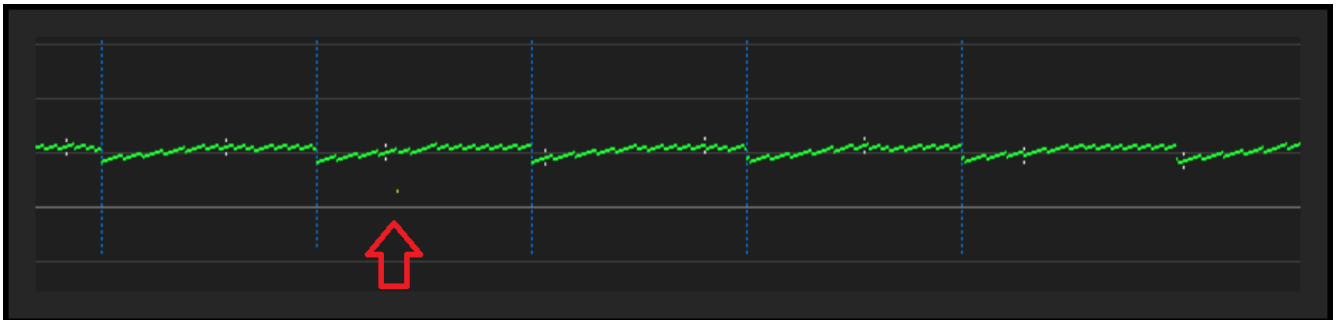
The green pixels indicate that a packet was received without issues. The distance between each green pixel and the brightest horizontal line indicates how much time was left when the packet was decoded, before a drop would have been caused. The thinner horizontal lines indicate seconds, so all the green packets here arrive between about 0.6 and 1.0 seconds before it's too late – which corresponds with a Delay setting of about 1 second.

Yellow pixels (1) indicate a packet that was restored using recovery packets (drawn in blue, 3). In this case a single packet was lost, so as soon as the first recovery packet has arrived it could be reconstructed and decoded. At that point, based on where the yellow dot is drawn, about 0.4 seconds of time were remaining for decoding it in time.

As you can see here, after a sequence of blue (recovery) packets, the green pixels have a bigger delay, which is caused by the rate limiter: Because it takes some time to send all the recovery packets, the packets for the next block of audio are delayed a bit. Increasing the rate limiter maximum speed will reduce these drops – but as described before, increasing it too much may cause dropouts.

White pixels above and below the green line (2) indicate keyframes.

If the decoder plays silence, the background of this display (at that point in time) is colored red.



If you see occurrences like this, you can see that the yellow packet was recovered only just in time, and that could be a reason to increase the Delay a bit. Or reduce the forward error correction span setting.

Hardware and stability

Most stations that run MicroMPX decoders run them on a Raspberry Pi 3 or 4. They are cheap, small and reliable. But, they can easily overheat, so if you place them in a building where it can get hot, some cooling might be needed. This is much less of an issue since MicroMPX version 4, which requires 50% less processing power.

Raspberry Pi + HifiBerry

If you want to use a Raspberry Pi, we recommend a Pi 3B+ or 4B, with HifiBerry sound card. That's also what our downloadable image is for, if you use that with a HifiBerry sound card it will run without the need to configure anything (unless you want to use the HifiBerry's inputs – then you need to edit the file /boot/config.txt and change the line dtoverlay=hifiberry-dac into dtoverlay=hifiberry-dacplusadc, or dtoverlay=hifiberry-dacplusadcpro if you have the DAC+ADC Pro sound card.)

At the moment of writing, Raspberry Pi MicroMPX decoders are running at hundreds of transmitter sites, and there have been very few issues. One radio group that we talk to regularly has been feeding 60 transmitters with Pi's for over a year now, and hasn't had a single issue with those yet.

MicroMPX+ for bitrates down to 176 kbit/s

MicroMPX+ is a different codec from MicroMPX, which also requires an extra license. It uses some tricks to lower the bitrate further. In that regard it's comparable with AAC+ vs AAC, or MP3Pro vs MP3, for example.

Alongside the audio, MicroMPX+ also transmits data that describes what type of artifacts have been created by the encoder, and the decoder then attempts to filter out those artifacts. This filtering changes the waveform, and can result in (small and short) overshoots, which require a clipper (which is included) to get rid of. Changes in the waveform may also affect the RF bandwidth that's used, which usually isn't a problem, but if you're using the Stokkemask (ITU recommendation ITU-R.SM1268) filter in Stereo Tool, or RF bandwidth control in BreakAway, BaOne, Omnia.7, 9 or 9sg, you can't rely on that it fully survives the codec anymore, especially at the lowest bitrates.[1]

1 So what does this mean? Reducing the RF bandwidth, which is a feature that only a few FM processors (listed above) offer, can slightly improve the reception, especially in fringe/multipath areas. MicroMPX+ will retain most of this effect, so it's still useful to turn it on.

In some countries (the Netherlands and Belgium), ITU-R.SM1268 compliance is mandatory – with processors that don't support it it's often achieved by modulating at a lower level or reducing stereo separation. In those countries, you probably don't want to use MicroMPX+, unless you do the same thing.

In the end, for listeners it shouldn't be possible to notice any difference if you switch to MicroMPX+ with lower bitrates.

MicroMPX+ Variable Bitrate for even lower bitrates
If you want to go even lower in bitrate, for example if you have to pay for the bandwidth that's actually being used instead, you can enable Variable Bitrate mode. This is especially useful for talk radio stations with moments of (near)-silence and mostly mono content.

Both MicroMPX and MicroMPX+ will always use all the available bandwidth to encode the incoming signal as precisely as possible, unless there is no information to encode anymore, which means that typically only pure silence causes the bitrate to drop below the configured bitrate. But since FM reception has a noise threshold, for (near) silence it doesn't really matter what the encoder sends. Aside from this, if the audio quality is good for stereo sounds, which requires encoding of both the L+R and (potentially asymmetrical) L-R parts of the band, it's safe to drop the bitrate considerably for mono content, which is easier to encode.

Variable Bitrate allows you to set a noise threshold level, thus increasing the chance that the bitrate drops below the configured value. This is somewhat similar to reducing the number of bits per sample, which is sometimes also used to reduce bitrates (with dithering). For music (except classical music), the effect will usually be very small, but for speech it can be quite big. Talk stations typically achieve average bitrates below 100 kbit/s, using a high enough VBR threshold at 176 kbit/s.

The artifacts created by the dithering of the audio are included in what the MicroMPX+ encoder reports to the decoder, so what comes out is filtered and sounds better than just dithering it would. The 19 kHz stereo pilot and RDS areas of the spectrum are not affected by the VBR noise threshold setting.

Appendix 1: Accessing MicroMPX from other software

It is possible to access almost all of the MicroMPX encoder and decoder settings, meters and logs remotely from other programs. This appendix describes how.

The basics

The MicroMPX encoder and decoder software can be controlled remotely via a HTTP interface. Besides though our own (proprietary, highly optimized) communication mechanism, most of the elements in the web interface can be accessed via JSON. This document only describes the JSON interface.

This description also applies to any other Thimeo software or hardware with a web interface, such as Thimeo Stereo Tool and Thimeo STX.

Please note that the JSON parser inside our software is very limited. Because our programs need to be able to run for years on end without any glitches in the audio, we had to write our own parser that - for example - never allocates memory. So, it works with the strings that we're showing in this document, for differently formatted strings it may work - or not.

Every parameter and meter in our software has a unique ID, which is an integer number. Where possible we have reused the numbers between different programs, so some of the ID's - for example for sound card settings - are shared between them. We can provide a list with ID's, but you can also look them up yourselves in the web interface – more about that below.

Normally, each web client that connects to the web interface is assigned a unique session ID. This is needed to make sure that each client displays the same information, and all changes are communicated to all the clients. The session ID is assigned on the first request that is sent, and needs to be included in all subsequent requests. Since this would make communication with JSON more difficult (it would require setting up a session, and keeping track of when a session is no longer valid), a special session ID -1 is available which assumes nothing about the state of the client. So for example, when a meter is read twice and the value didn't change, with ID -1 the value will be sent to he client both times - for any other session ID, the second request will be ignored if nothing changed. Unless data usage is important (and if it is, you shouldn't be using JSON!), you can just use ID -1.

Some examples.

This is probably easiest explained using some examples.

```
http://127.0.0.1:8080/json-1/list{"10172":{}}
```

or

```
list
{
    "10172" : {}
}
```

This talks to the MicroMPX on port 8080 of localhost, and gets JSON data for session ID -1 (hence json-1). It requests data for widget ID 10172 (current output level) without specifying anything. The response looks like this:

```
{
    "10172":
    {
        "enabled": "1",
        "value": "0.827557",
        "value_db": "-1.644043"
```

```
        }
    }
```

Which means that the meter is enabled (it would be disabled if the MicroMPX decoder isn't running), and the level is -1.644043 dB (0.827557 in absolute value).

Some more interesting data is available under 11714 and 11715:

```
    list
    {
        "11714" : {},
        "11715" : {}
    }
```

This returns the stream status:

```
    {
        "11714":
        {
            "enabled": "1",
            "value": "0",
            "text": "Feedback: Stream status|1"
        }
        "11715":
        {
            "enabled": "1",
            "value": "1",
            "text": "Feedback: Playing now|1"
        }
    }
```

Value 1 for 11714 means that the stream is ok. 2 means error - for example packet loss. 0 means the value has never been initialized. For 11715, 1 means everything is ok, 2 means playing either silence or the backup player is active. 0 again means that the value has never been set.

You can change values and put multiple values in a single request. For example:

```
{
    "1000" :
    {
        "forced" : "1",
        "new_value" : "55"
    },
    "1001" : {},
    "6": {}
}
```

Which means something like this:

- Set parameter 1000 to 55
- Give me the value of parameter 1000, forced, even if it didn't change. "forced" makes JSON *always* send you the return value, even if it hasn't changed. With session ID -1 that doesn't really matter.
- Give me the value of parameter 1001
- Give me the value of parameter 6

```
http://127.0.0.1:8080/json-1/%7B%221000%22:%7B%22forced
%22:%221%22,%20%22new_value%22:%2255%22%7D,%20%221001%22:%20%7B%7D,
%20%226%22:%20%7B%7D%7D
```

Return string:

```
  {
      "1000" :
      {
          "enabled" : "0",
          "value" : "55.000000",
          "text" : "Slope to 9|55 dB/oct"
      },
      "1001" : "ERROR_UNKNOWN_ID",
      "6":
      {
          "enabled" : "1",
          "value" : "1.000000",
          "text" : "Pre amplifier| 0.00 dB (100.0%)"
      }
  }
```

"enabled" means that the widget (slider in this case) is enabled in the GUI.
"value" = raw value (always a float number in this case, for float arguments)
"text" = text in GUI (before the | is the label, after is the value).

1001 didn't exist so you get an "ERROR_UNKNOWN_ID".

To retrieve current settings for the "Volume (MPX level)":

```
http://127.0.0.1:8080/json-1/list{"153":{}}
{ "153": { "enabled": "1", "value": "1.000000", "text": "Volume (MPX
level)| 0.00 dB (100.0%)" } }
```

Too change the volume:

```
http://127.0.0.1:8080/json-1/list{"153":{"forced":"1","new_value":"0.5"}}
```

You can also get a list of error messages, on ID 11086. The reply looks like this:

```
  {
      "11086":
      {
          "0":
          {
              "count": 1,
              "text": "2019/04/01 08:06:51 You're using a legacy
                       MicroMPX stream format. Update all encoders
                       and decoders and then switch to Version 2
                       for improved codec behavior."
          }
      }
  }
```

But you might also get a longer list (indentation removed to save some space):

```
{ "11086": { 0: { "count": 1, "text": "2019/06/21 16:54:40 License is not
recognized as a valid MicroMPX license." }, 1: { "count": 1, "text":
"2019/06/21 16:54:40 You do not have a license for MicroMPX decoder!" }, 2:
```

{ "count": 1, "text": "2019/06/21 16:54:57 Audio loss\nPackets arrived too late\n1.102 Seconds of silence" }, 3: { "count": 1, "text": "2019/06/21 16:54:57 Audio loss\nPackets arrived too late\n0.000 Seconds of silence" }, 4: { "count": 1, "text": "2019/06/21 16:54:57 Audio loss\nPackets arrived too late\n0.004 Seconds of silence" }, 5: { "count": 1, "text": "2019/06/21 16:54:57 Audio loss\nPackets arrived too late\n0.007 Seconds of silence" }, 6: { "count": 1, "text": "2019/06/21 16:54:57 Audio loss\nPackets arrived too late\n0.085 Seconds of silence" }, 7: { "count": 1, "text": "2019/06/21 16:55:01 Buffer out of range (84.64% / 1866ms full) - resetting" }, 8: { "count": 1, "text": "2019/06/21 16:55:21 Audio loss\nPackets arrived too late\n2.113 Seconds of silence" }, 9: { "count": 1, "text": "2019/06/21 16:55:21 Audio loss\nPackets arrived too late\n0.010 Seconds of silence" } } }

To clean an error, you can set "new_value" to the error index (0, 2, 3, 4, 5, 6, 7, 8, 9 in the example above) to clear an error. -1 clears all errors. Count indicates how often the error occurred, in this example it's always 1 but it can be higher. As things are now, the maximum number of errors that can be reported is 10, so you need to regularly clear them to not miss any messages.

O, and yes: An error message can contain enters!

Clicking on a button is possible by setting the value:
`http://127.0.0.1:8084/json-1/list%7B%2211285%22:%7B%22forced%22:%221%22,%20%22new_value%22:%221%22%7D%7D`

Here's a longer example output string, to give you an impression of what's possible:

{ "4646": { "enabled": "1", "value": "1", "text": "MPX Enabled|YES" }, "4647": { "enabled": "1", "value": "8854", "text": "Port|8854" }, "4648": { "enabled": "1", "value": "1.000000", "text": "Delay|1.000 sec" }, "5889": { "enabled": "1", "value": "", "text": "Subscribe to multicast IP|" }, "4649": { "enabled": "1", "value": "", "text": "Backup file|" }, "4650": { "enabled": "0", "value": "60.000000", "text": "Backup timeout|60 sec" }, "153": { "enabled": "1", "value": "1.000000", "text": "Volume (MPX level)|0.00 dB (100.0%)" }, "461": { "enabled": "1", "value": "0", "text": "Correction enabled|NO" }, "462": { "enabled": "0", "value": "5.000000", "text": "RC|DISABLED" }, "5256": { "enabled": "1", "value": "0", "text": "High frequency dropoff|NO" }, "5257": { "enabled": "0", "value": "0.000000", "text": "RC|0.000" }, "10174": { "enabled": "1", "value": "0", "text": "Generate test tone|NO" }, "10175": { "enabled": "0", "value": "Sine", "text": "|Sine" }, "10176": { "enabled": "0", "value": "400.000000", "text": "Frequency| 400.0 Hz" }, "5349": { "enabled": "0", "value": "0", "text": "Password protect stream|NO" }, "5350": { "enabled": "0", "value": "", "text": "Active hash|" }, "11284": { "enabled": "1", "value": "", "text": "New password|" }, "11286": { "enabled": "1", "value": "", "text": "New hash|" }, "452": "ERROR_UNKNOWN_ID", "11159": { "enabled": "1", "total_count": "1024", "keyframes_received": "2", "packets_received": "1024", "packets_recovered": "0", "played_silence": "0" }, "11714": { "enabled": "1", "value": "0", "text": "Feedback: Stream status|0" }, "11715": { "enabled": "1", "value": "0", "text": "Feedback: Playing now|0" }, "11086": { 0: { "count": 1, "text": "2019/04/01 08:06:51 You're using a legacy MicroMPX stream format. Update all encoders and decoders and then switch to Version 2 for improved codec behavior." } }, "11078":

{ "enabled": "1", "value": "2.1.1", "text": "SWVER|2.1.1" }, "10252":
{ "enabled": "1", "value": "0.997710", "value_db": "-0.019914" } }

List of ID's.

Here are some of the ID's for MicroMPX.

```
License:
NOSAVEPARAM_Request             11087 (string) Request code
NOSAVEPARAM_Key                 11088 (string) License key
NOSAVEPARAM_ACTIVATE_REGKEY     10544 (button)
NOSAVEPARAM_REG_LicensedTo      10534 (string) license ID
NOSAVEPARAM_REG_LicensedTill    11092 (string) License valid till

Sound card:
NOSAVEPARAM_DSO_Device           9048 (enum)   MPX output
PARAM_DSO_Volume                  153 (float)  MPX output level
PARAM_fmoutputtilt_tone_on      10174 (bool)   Generate test tone
PARAM_fmoutputtilt_tone_squar   10175 (enum)   Test tone type
PARAM_fmoutputtilt_tone_freq    10176 (float)  Test tone frequency

Tilt correction settings:
PARAM_fmoutputtilt_on
PARAM_fmoutputtilt_rc             458
PARAM_fm_calibrate_hightilt_on
PARAM_fm_calibrate_hightilt_rc

Encoder main settings:
Input level correction            452 Input level gain
PARAM_UMPX_RedundancyPacketDelay 4588 Error correction size/Delay
PARAM_UMPX_RedundancyPacketOverhead 4589 Error correction Overhead
PARAM_UMPX_TRPMaxBytesPerSecond_Enabled 4596 Rate limiter on/off
PARAM_UMPX_TRPMaxBytesPerSecond 4596 Rate limiter speed
PARAM_UMPX_Bitrate_V0           4016 Bitrate for MicroMPX V1
PARAM_UMPX_Bitrate_V1           6682 Bitrate for MicroMPX V2/V3
PARAM_UMPX_Enabled_0            4217+ Enable stream (0-99)
PARAM_UMPX_IP_0                 4317+ Stream URL (0-99)
PARAM_UMPX_Port_0               4417+ Stream Port (0-99)
PARAM_UMPX_Interface_0          4653+ Stream NIC (0-99)

Decoder main settings:
PARAM_UMPX_Decoder_Enabled      4646 (bool)
PARAM_UMPX_Decoder_Port         4647 (int)
PARAM_UMPX_Decoder_SubscribeMulticastIP 5889 (string)
PARAM_UMPX_Decoder_Delay        4648 (float)
PARAM_UMPX_Decoder_BackupFile   4649 (string)
PARAM_UMPX_Decoder_BackupTimeou 4650 (float)

Security:
PARAM_UMPX_PasswordProtectStre  5349 (bool)
PARAM_UMPX_Key                  5350 (string)
NOSAVEPARAM_UMPX_Password      11284 (string)
NOSAVEPARAM_UMPX_Password_Appl 11285 (button)
NOSAVEPARAM_UMPX_Hash          11286 (string)
NOSAVEPARAM_UMPX_Hash_Apply    11287 (button)
```

```
Errors:
ID_THIMEO_KOEK_ERRORS          11086 (list of errors)

Software version number:
NOSAVEPARAM_SWVERSION          11078 (string)
```

Note: This list is not complete, please ask if you need more settings. Nearly anthing that's available in the web interface can be set via JSON.

Other ID's

The list above is probably not complete; new ID's are being added regularly, and for example Stereo Tool has thousands of ID's, which makes it impossible to list them all.

If you want to access a specific value, you can find the ID in the web interface. Just use the developer features of your web browser to investigate a slider or meter – the ID's are visible in the DOM. Most of them will be accessible via JSON (depending on the widget type).

Appendix 2: Possible problems and solutions

This section describes some issues that you might run into, plus how to solve them.

Peak control is not perfect
Make sure that the input level and gain on the encoder are set such that peaks in the input signal reach exactly 0 dB (100%) on the waveform display. A warning will be shown if the level is too low. Note that setting it too high is bad as well, because values above 100% cannot be encoded, so the level must match exactly.

Audio dropouts
If you have audio dropouts, the first thing to do is to make sure that you have a reliable connection between encoder and decoder. But since that's not always possible, the next best thing is to enable Forward Error Correction. See section “MicroMPX Encoder configuration: Forward error correction**”.**

Decoded RDS level is too low and has bitstream errors at regular intervals
According to the RDS specification, the RDS signal must be synchronized to the pilot. Processors with a built-in RDS encoder won't have a problem with this, but if you're using an external RDS encoder that's feeding into a processor, there's usually a connector on that RDS encoder which you can feed the 19 kHz pilot tone into, which the RDS encoder then uses to synchronize to. If the RDS signal isn't synchronized, or even drifts relative to the pilot frequency, RDS reception might not work on some receivers, and MicroMPX won't be able to properly transmit it.

Appendix 3: Ancillary data (PROPOSAL – V1.1)

This appendix is aimed at hardware manufacturers and users of our MicroMPX libraries.

A MicroMPX stream can carry blocks of extra or "ancillary" data. This data is transported as part of the existing stream, and can be read by the decoder. From the MicroMPX library's perspective, it's just a data channel, and it's up to individual users of the library to decide how to use it.

To prevent ancillary data from one vendor to be misinterpreted by another, the ancillary data block must start with a 2-byte vendor identifier, typically the first 2 characters in the company name – for Thimeo it would be 'T' 'h'. To avoid conflicts with other vendors' identifiers, please contact us to get a unique code if you plan to use this.

As an example, Thimeo-specific ancillary data would look like this:

Byte	0	1	2	3	4	5	6
Value	'T'	'h'	XX	XX	XX	XX	XX

Ancillary data and timing
There are 2 ways to schedule the sending of ancillary data:

- With the next keyframe. Timing varies based on keyframe frequency.
- Immediately. This forces insertion of an extra keyframe, which causes slightly more overhead.

Refer to the library function umpxEncoderSendAncillaryData for more info.

The decoder knows at which audio time stamp the ancillary data was sent.

Ancillary data is sent only once. For certain types of persistent data such as the current state of GPO pins, we recommend to send it once using "immediately" at the moment the value changes, and then repeat it at a lower rate using "next keyframe" scheduling, in case the decoder missed the message or is rebooted for example.

Vendor-independent ancillary data
Sometimes it's necessary to send standardized data between devices made by different vendors.

To support this, and allow combining vendor-independent with vendor-dependent data, the following encoding shall be used.

Instead of the aforementioned 2-byte vendor-specific codes, a single byte is used with the highest bit set (0x80). The remaining 7 bits indicate the type of data.

To efficiently support both extremely short and very long messages, we have set the following ranges:

- 0x80 – 0xBF has 1 extra byte indicating the size of the packet.
- 0xC0 – 0xDF has only 1 byte of data.
- 0xE0 – 0xFF has 2 extra bytes indicating the size of the packet.

Examples:

Byte	0	1	2	3	4	5	6
Value	0x87	0x05	XX (1)	XX (2)	XX (3)	XX (4)	XX (5)

0x81, 5 bytes follow, plus 5 bytes.

Byte	0	1
Value	0xCE	XX

0xCE, plus 1 byte

Byte	0	1	2	3	4	5	6	…	562
Value	0xF1	0x02	0x30	XX (1)	XX (2)	XX (3)	XX (4)		XX (560)

0xF1, 0x0230 bytes follow (big endian), followed by 560 bytes of data.

Multiple blocks of vendor-independent ancillary data can be concatenated, and vendor-specific data can be added at the end:

Byte	0	1	2	3	4	5	6	7	8
Value	0x81	2	XX	XX	0xCE	XX	‘T’	‘h’	XX

In short: When reading through the data on the decoder end, if you encounter a byte in the range 0x80-0xFF, you can either interpret it or skip the appropriate number of bytes and check what comes next. If you encounter a 2-byte code that doesn’t start with 0x80-0xFF, if it’s not your own vendor code you should just ignore the rest of the data.

GPO data
At this moment, the only type of vendor-independent ancillary data that is defined is GPO data. It shall be encoded as follows:

- 0x81
  GPO data follows.
- Size
  The number of bytes with GPO data.
  This will typically be 1 or 2 bytes, for up to 8 or up to 16 bits of GPO data.
- XX – first 8 bits. Highest bit (0x80) is GPO pin 0, 0x40 is GPO pin 1, etc.
- XX – if size > 1, next 8 bits. Highest bit is GPO pin 8, etc.

Example:

Byte	0	1	2
Value	0x81	0x01	0xB0

GPO data, 1 byte, GPO pins 0 and 1 are set, any others are not.

If you need different vendor-independent ancillary data
In principle, given the rules above, any group of vendors can define new messages. But we do need to keep track of which codes are in use, and there’s a limit to the number of codes that can be used, so please contact us to discuss it. If applicable, we will add your code to this document.