

Softwarebasierte Realisierung eines Funkmeldeempfängers / Fünftonfolgendecoders mittels Javascript

Eine Hausarbeit von:

David Breidert

Talstraße 33

07545 Gera

david.breidert@me.com

Eingereicht bei:

Thüringer Landesfeuerwehr- und
Katastrophenschutzschule

Silbitzer Weg 6

07586 Bad Köstritz

Inhaltsverzeichnis

1 Einleitung.....	3
1.1 Die analoge Alarmierung.....	3
1.1.1 Die Fünftonfolge.....	4
1.1.2 Vor- und Nachteile der analogen Alarmierung.....	5
2 Das Projekt: ZVEI-Decoder.....	7
2.1 Ziele.....	7
2.1.1 Funktionen.....	7
2.2 Auswahl der Programmiersprache.....	7
3 Analyse des Audiosignals.....	9
3.1 Grundlagen.....	9
3.2 Abtasten des analogen Signals.....	9
3.2.1 Ansatz 1: Autokorrelation.....	10
3.2.2 Ansatz 2: Frequenzspektrumanalyse.....	11
4 Umsetzung in Javascript.....	12
4.1 Die Benutzeroberfläche.....	12
4.2 Funktion mounted.....	13
4.3 Funktion updateData.....	14
4.4 audio.js.....	16
4.4.1 Funktion getUserAudio.....	16
4.4.2 Funktion createContext.....	16
4.4.3 Funktion getCurrentFrequencyFft.....	17
4.5 zvei.js.....	18
4.5.1 Funktion getTonNummer.....	18
4.5.2 Funktion getValidatedTonfolge.....	18
4.6 Die Einsatzmitteldatenbank.....	20
5 Anhang.....	21

5.1Bedienungsanleitung.....	21
5.1.1Links.....	21
5.1.2Benötigte Hardware.....	21
5.1.3Installation.....	21
5.1.4Nötige Windows-Einstellungen.....	21
5.1.5Nach dem Programmstart.....	22
5.1.6Dashboard.....	24
5.1.7 Einsatzmitteldatenbank.....	25

1 Einleitung

In Thüringen gibt es 1.609 freiwillige Feuerwehren mit insgesamt ca. 34.000 aktiven Mitgliedern in den Einsatzabteilungen ([1]). Während in den größeren Städten Berufsfeuerwehren 24 Stunden und 7 Tage die Woche auf der Feuerwache in Einsatzbereitschaft sind, sind die freiwilligen Kräfte vor allem im ländlichen Bereich essentiell, um den Brandschutz und die dringliche technische Hilfe sicherzustellen. Diese Hausarbeit beschäftigt sich primär mit der Alarmierung dieser freiwilligen Kräfte.

1.1 Die analoge Alarmierung

Die Alarmierung von Feuerwehren und Rettungsdiensten in Thüringen erfolgt flächendeckend noch über den 4m-Analogfunk, also im gleichen Netz in dem auch der Sprechfunkverkehr stattfindet. Alle Teilnehmer dieses Netzes können sich gegenseitig hören und sich untereinander verständigen.

Da auf dem selben Kanal hunderte verschiedene Kräfte alarmiert werden, ist es aufgrund der niedrigen Einsatzfrequenz einzelner Feuerwehren / Rettungsdienste – im Vergleich mit der Gesamteinsatzfrequenz – unrealistisch, dass jeder Feuer-



Abb. 1: typischer Funkmeldeempfänger

(Quelle: https://www.gfd-katalog.com/ludwig_feuerschutz/quattro-xlixisi/quattro-xli/25597)

wehrmann bzw. Rettungsdienstmitarbeiter stets ein Funkgerät mit sich trägt und darauf wartet angesprochen zu werden. Deshalb wurde ein alternatives System entwickelt, mit dem die Einsatzkräfte auf eine Alarmierung aufmerksam gemacht werden:

Einsatzkräfte größerer freiwilliger Feuerwehren sowie von Berufsfeuerwehren und Rettungsdiensten tragen für die Alarmierung meist einen sogenannten Funkmeldeempfänger (FME) mit sich. Die Funkmeldeempfänger werden im Einsatzfall von der alarmierenden Stelle – meist ist dies die zuständige zentrale Leitstelle – ausgelöst. Hierzu wird eine, auf dem genutzten Funkkanal einzigartige, Tonfolge in den Funk eingespielt, die auf den entsprechenden Funkmeldeempfängern programmiert ist. Bei Erkennen einer programmierten Tonfolge gibt der Melder ein Alarmsignal und schaltet im Anschluss den Funk laut, woraufhin der Disponent eine Alarmdurchsage mit den Einsatzdaten durchsagen kann.

Da die Anschaffung und Programmierung von Funkmeldeempfängern ein großer Kostenfaktor sein kann, wählen kleinere Feuerwehren oft eine andere Variante, um die Alarmierung der eigenen Kräfte sicherzustellen: die Sirenenalarmierung.

Bei der Sirenenalarmierung wird von der alarmierenden Stelle statt den Funkmeldeempfängern eine Sirene ausgelöst, die im betroffenen Ort zentral platziert ist, um möglichst viele Einsatzkräfte zu erreichen.

1.1.1 Die Fünftonfolge

Egal ob Sirene oder Funkmeldeempfänger, die Auslösung des Alarms basiert auf einem deutschlandweit einheitlichen System.

Im Alarmfall spielt die alarmierende Stelle in den Funkkanal, auf den die Funkmeldeempfänger und/oder Sirenen programmiert sind, sogenannte Fünftonfolgen ein. Fünftonfolgen sind – wie der Name schon sagt – eine Abfolge von fünf einzelnen Tönen, die von dem jeweiligen Empfangsgerät erkannt werden.

Der Aufbau dieser Tonfolge ist deutschlandweit einheitlich und in einer technischen Richtlinie¹ festgelegt worden. Unter ist in dieser Richtlinie zu finden welche Frequenzen (siehe Abb. 2) sowie Dauer die einzelnen Töne der Tonfolge haben sollen, wie oft diese wiederholt werden soll, und welche zeitlichen Abstände zwi-

¹ <https://www.lfs-bw.de/Fachthemen/Digitalfunk-Funk/Documents/Pruefstelle/TRBOS-Funkalarmierung.pdf>

schen mehreren hintereinander eingespielten Tonfolgen eingehalten werden sollen (siehe Abb. 3). Es wird auf zehn verschiedene Frequenzen (Ziffer 0-9) und eine Ausweichfrequenz für sich wiederholende Töne (Ziffer R) zurückgegriffen. Diese Frequenzen richten sich in Deutschland nach dem ZVEI-Standard.

Ziffer	1	2	3	4	5	6	7	8	9	0	R
Tonhöhe/Hz	1060	1160	1270	1400	1530	1670	1830	2000	2200	2400	2600

Abb. 2: Tonfrequenzen, aus denen die Fünftonfolgen aufgebaut werden

(Quelle: <https://de.wikipedia.org/wiki/5-Ton-Folge>)

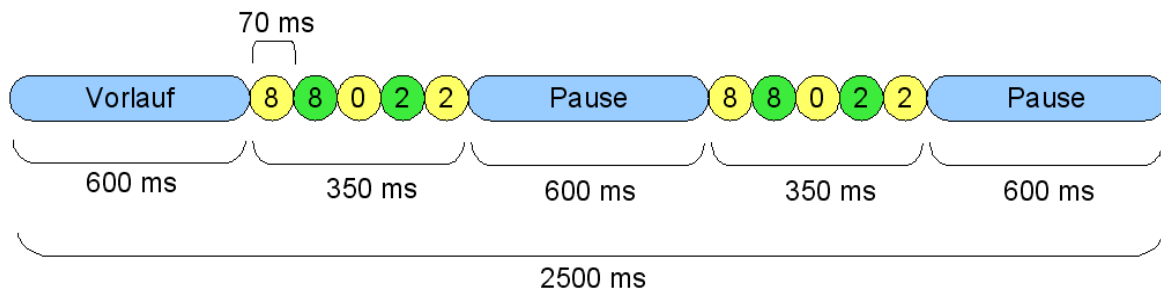


Abb. 3: zeitlicher Ablauf einer Funkalarmierung

Quelle: <https://de.wikipedia.org/wiki/5-Ton-Folge>

So würde sich die Tonfolge 88910 zum Beispiel folgendermaßen aufbauen:

2000Hz – 2600Hz (da die Ziffer „8“ hier auf eine weitere Ziffer „8“ folgt wird die Reservefrequenz codiert) – 2200Hz – 1060Hz – 2400Hz.

Dass diese Parameter einheitlich festgelegt sind ist wichtig, damit die empfangenden Geräte die Tonfolgen zuverlässig erkennen können.

1.1.2 Vor- und Nachteile der analogen Alarmierung

Die analoge Alarmierung bietet einige Vorteile, aber auch Nachteile gegenüber digitalen Alternativen, wie etwa eine auf POGSAC oder TETRA basierende Variante.

Einer der größten Nachteile der analogen Alarmierung – und auch der Grund warum deutschlandweit der Trend immer mehr in Richtung digitale Alarmierung geht – ist die fehlende Verschlüsselung. Theoretisch kann jeder mit einem Funkgerät und dem nötigen technischen Fachwissen den Funk und somit auch die Alarmierung abgreifen.

Weitere Nachteile sind die Überlastung des Funkkanals durch lange Sprachdurchsagen (Jede Alarmierung belegt den Funkkanal für ca. 20-30 Sekunden), sowie

der fehlende Feldstärkealarm bei analogen Funkmeldeempfängern, wodurch beim Aufhalten in einem Funkloch eine Alarmierung verpasst werden kann.

Größter Vorteil gegenüber den digitalen Alternativen stellt die Alarmierungsgeschwindigkeit dar. Vom Auslösen des Alarms in der Leitstelle bis zum Ertönen des Funkmeldeempfängers vergeht meistens nicht mehr als eine Sekunde.

2 Das Projekt: ZVEI-Decoder

In diesem Kapitel soll mithilfe von Quellcodeauszügen die Arbeitsabläufe des Programms erklärt werden, die zur Erkennung und Ausgabe der Fünftönfolgen führen

2.1 Ziele

Ziel des Programms, dass in dieser Projektarbeit entwickelt wird ist die softwarebasierte Decodierung von Fünftönfolgen. Später soll auf diese Funktionalität mit zusätzlichen Modulen / Funktionalitäten aufgebaut werden, um Einsatzkräfte in ihrer Arbeit zu unterstützen. Diese Funktionen können von einer SMS-Alarmierung, bis zu einer Audio-Aufzeichnung der Sprachdurchsage reichen.

Um diese Zusatzfunktionen im Falle einer Alarmierung entwickeln zu können ist es zwingend notwendig den Alarmfall zügig und zuverlässig erkennen zu können. Da Feuerwehren – wie in der Einleitung beschrieben – flächendeckend über Fünftönfolgen alarmiert werden, ist die Decodierung dieser zwingend notwendig um später darauf aufbauen zu können.

2.1.1 Funktionen

Folgende Funktionen sollen im Rahmen der Entwicklung realisiert werden:

- zuverlässige Decodierung der Fünftönfolgen eines Funkkanals über einen Funkscanner.
- Visuelle Darstellung der registrierten Fünftönfolgen und des Zeitpunkts der Registrierung.
- Verknüpfung von Fünftönfolgen mit einer Datenbank von Einsatzmitteln, die im Programm bearbeitet werden kann.

2.2 Auswahl der Programmiersprache

Vor Beginn des Projekts galt es die Entscheidung zu Treffen, welche Programmiersprache verwendet wird. Aufgrund meiner persönlichen Erfahrung in der Web-

entwicklung entschied ich mich für einen unorthodoxen Weg und wählte JavaScript als alleinige Programmiersprache für ZVEI-Decoder.

JavaScript ist ursprünglich als Programmiersprache von Webbrowsern entstanden um einfache Skripte für Webseiten zu entwickeln. Mit der Zeit wurde das Potential von JavaScript immer größer. Heutzutage ist es möglich JavaScript Apps zu entwickeln und diese mittels Zusatzpaketen als eigenständige Programme zu packen.

ZVEI-Decoder nutzt folgende Technologien:

- ElectronJS (<https://www.electronjs.org>), für das Packen der Web-App als eigenständige Anwendung
- VueJS (<https://vuejs.org>), als JavaScript-Framework
- VuetifyJS (<https://vuetifyjs.com>), als User-Interface Bibliothek

Einen kompletten Webbrowser als App zu verpacken hat natürlich gegenüber anderen Alternativen sowohl Nach- als auch Vorteile:

Durch die zwischengeschaltete JavaScript-Engine ist die Ausführungs- und Ausführungsgeschwindigkeit vieler Prozesse langsamer als bei einer hardwarenäheren Programmiersprache wie z.B. C++.

ZVEI-Decoder muss, um seine Funktion zu erfüllen eine Echtzeitanalyse eines Audiostreams vornehmen, was bei mangelnder Leistung der Engine oder des Rechners zu Problemen führen kann.

Aus diesem Grund wurden vor dem endgültigen realisieren des Projekts einige Tests meinerseits durchgeführt, um die Leistungsfähigkeit dieser Umgebung zu testen. Das Ergebnis dieser Tests lässt vermuten, dass die Leistung der JavaScript-Engine auf handelsüblichen PCs völlig ausreichend ist um eine solche Echtzeitanalyse durchzuführen.

Somit konnte die vermutete Einschränkung durch die Programmiersprache nicht bestätigt werden und das Vorhaben ungehindert weiter durchgeführt werden.

Ein großer Vorteil der gewählten Programmierumgebung ist hingegen die Portierbarkeit. Das fertige Programm kann in kürzester Zeit von einer Windowsanwendung auch als Macanwendung, oder sogar als Webanwendung umgeschrieben werden. Somit sind Nutzer nicht an ein Betriebssystem gebunden.

3 Analyse des Audiosignals

3.1 Grundlagen

In den Funk eingesprochene Sprache oder eingespielte Töne sind analoge Signale, welche mittels Frequenzmodulation auf eine Trägerfrequenz moduliert werden, damit diese übertragen werden können. Der Empfänger des Signals demoduliert im Anschluss die sogenannte Niederfrequenz von der Trägerfrequenz, um diese dann z.B. über einen Lautsprecher auszugeben.

Um das analoge Funksignal mittels Software auswerten zu können, muss dieses in ein digitales Signal umgewandelt werden (Analog / Digital – Umsetzung).

Dabei wird ein zeit- und wertkontinuierliches Signal in ein zeit- und wertdiskretes Signal umgewandelt ([2])

3.2 Abtasten des analogen Signals

Um digitale Werte aus dem zeit- und wertkontinuierlichen, analogen Signal zu entnehmen, muss dieses abgetastet werden.

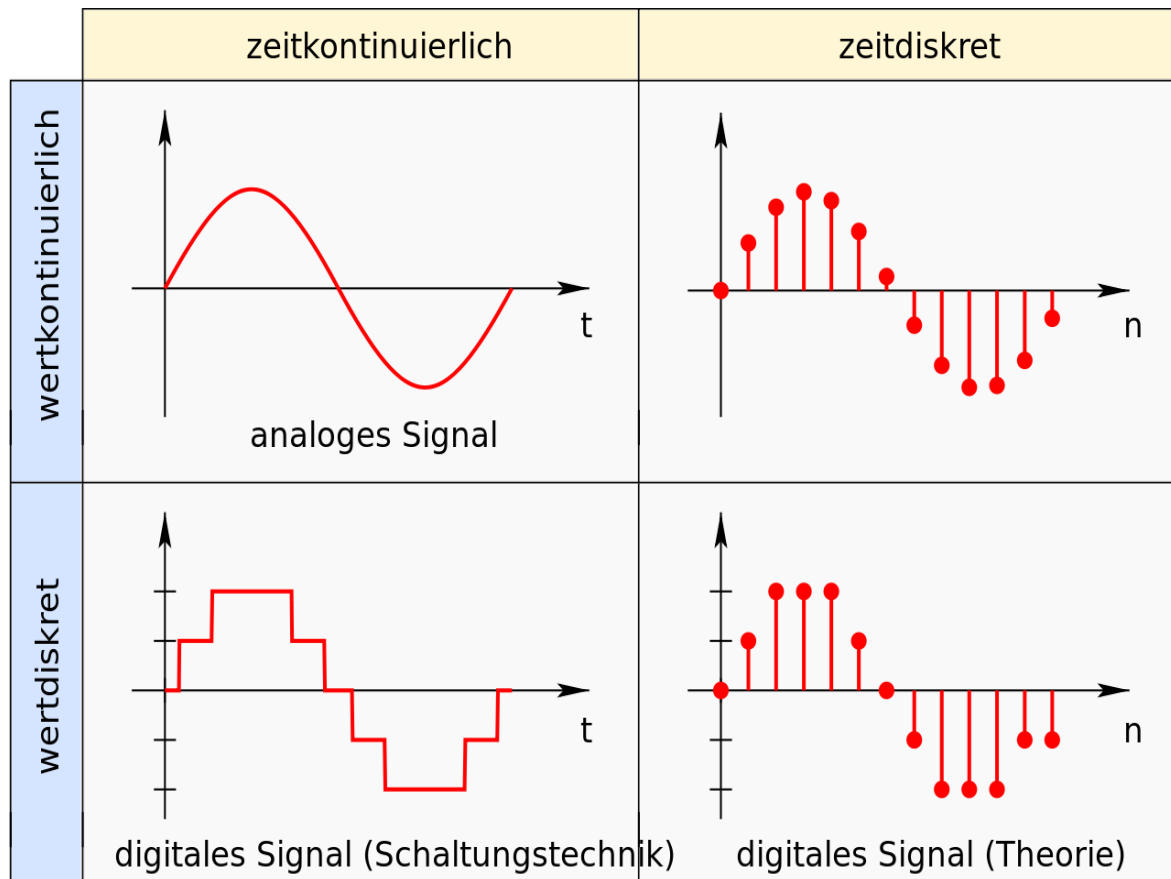


Abb. 4: Gegenüberstellung von analogen und digitalen Signalen

(Quelle:

https://upload.wikimedia.org/wikipedia/commons/f/f2/%C3%9Cbersicht_kontinuierliche_und_diskrete_Signale.svg)

Das Abtasten des Signals übernimmt die Soundkarte des Rechners, ZVEI-Decoder greift diese Werte mittels der nativen Web-Audio-API([3]) von JavaScript ab.

Diese Werte müssen nun weiter analysiert werden, um die spezifischen Frequenzen einer Ruftonfolge zu erkennen. Dafür gibt es zwei wesentliche Ansätze.

3.2.1 Ansatz 1: Autokorrelation

Autokorrelation ist der Vergleich eines Signals mit sich selbst zu einem früheren Zeitpunkt([4]).

Mit diesem Verfahren ist es möglich die Frequenz eines periodischen Signals (z.B. eines Analogfunktions) zu einem gewissen Zeitpunkt zu ermitteln. Dies geschieht – vereinfacht gesagt – indem man das Signal solange mit sich selbst zu immer später zurückliegenden Zeitpunkten vergleicht, bis es wieder annähernd gleich ist.

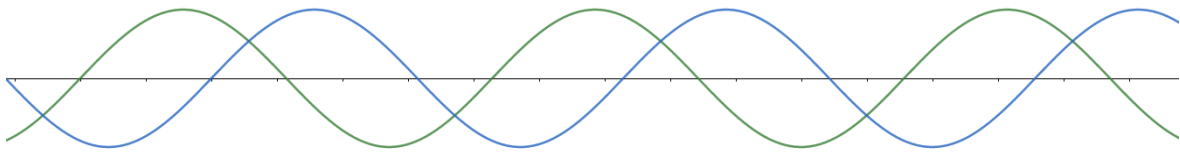


Abb. 5: vereinfachte graphische Darstellung einer Autokorrelation. Der blaue Graph wird so lange nach hinten verschoben, bis er über dem grünen liegt.

Hat man den Punkt erreicht, an dem das Signal dem Referenzsignal gleicht, kann man daraus auf die Periodendauer schließen, ergibt sich aus der Differenz des Zeitpunkts des Referenzsignals und des übereinstimmenden Signals. Die Periodendauer ist dann, im Falle eines Audiosignals, gleich der Frequenz in Hertz.

Dieser Ansatz eignet sich sehr gut für sehr saubere Signale, weniger jedoch für häufig stöbelastete Signale wie den Analogfunk, da überlagernde Störfrequenzen den Algorithmus schnell an seine Grenzen bringen und so Frequenzen verfälscht werden. Für die gewünschte Anwendung ist dieser Ansatz dementsprechend nicht geeignet.

3.2.2 Ansatz 2: Frequenzspektrumanalyse

Anders als bei der Autokorrelation wird bei der Frequenzspektrumanalyse nicht nur eine Frequenz ermittelt, sondern das Audiosignal wird in seine Einzelfrequenzen zerlegt. Störbehaftete Signale (wie etwa durch Rauschen) stellen hier nicht so ein großes Problem wie bei der Autokorrelation dar, da das gesuchte Signal vom Störsignal getrennt werden kann.

Die Berechnung eines Frequenzspektrums ist sehr rechenintensiv. Aufgrund moderner Computer und optimierten Algorithmen ist es heutzutage jedoch möglich eine Echtzeitberechnung des Frequenzspektrums durchzuführen.

Der am besten geeignete Algorithmus für eine Echtzeitberechnung des Frequenzspektrums eines Audiosignal ist die Fast-Fourier-Transformation (FFT)([5]), die Erklärung dieses Algorithmus würde den Umfang dieser Projektarbeit jedoch übersteigen.

Durch den gewaltigen Vorteil der niedrigeren Störanfälligkeit ist dieser Ansatz der geeignetste für ZVEI-Decoder.

4 Umsetzung in Javascript

Da die Programmierumgebung und die Methode zur Auswertung des Audiosignals nun geklärt sind, geht es in diesem Kapitel um die praktische Umsetzung des Projekts.

4.1 Die Benutzeroberfläche

Wie in Kapitel 2.3 bereits erwähnt wird die Benutzeroberfläche mithilfe der JavaScript Frameworks VueJS und VuetifyJs realisiert.

Neben einer „Dashboard“-Ansicht, die neben der Liste mit den erkannten Tonfolgen noch eine digitale Uhr und einen Aktivitätsmonitor für den Audiostream beinhaltet, gibt es noch eine zweite Ansicht zur Bearbeitung der Datenbank.

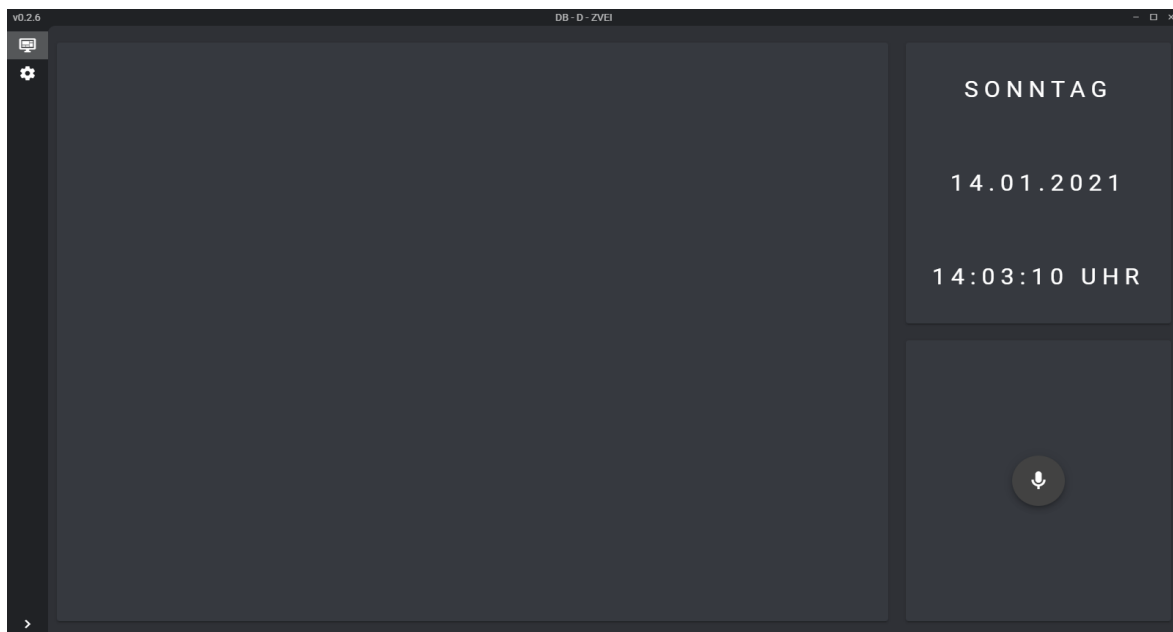


Abb. 6: Dashboard-Ansicht von ZVEI-Decoder

Umrandet werden diese beiden Ansichten von einer Kopf- und einer Seitenleiste, in der Kopfleiste finden sich Informationen zu Name und Version des Programms, sowie die Fenstersteuerung.

In der Seitenleiste lässt sich zwischen den beiden Hauptansichten umschalten.

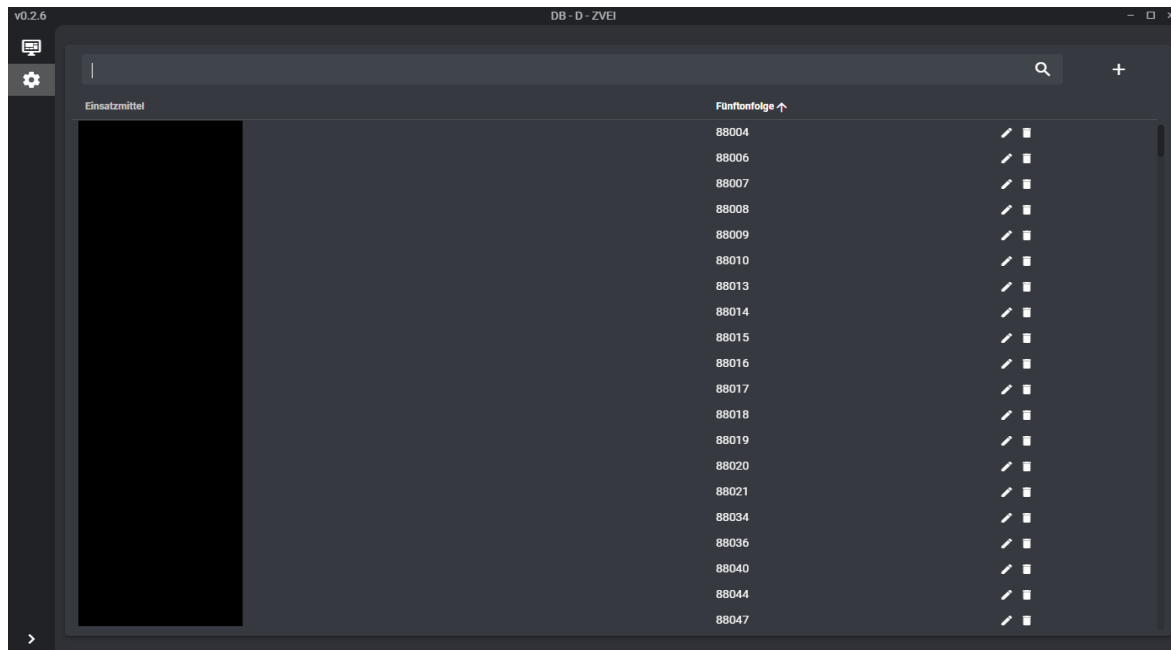


Abb. 7: Datenbankansicht von ZVEI-Decoder mit geschwärzten Einsatzmitteln

4.2 Funktion mounted

Die Funktion mounted ist der Einstiegspunkt des Programms. Sie hat folgende Aufgaben:

- Einrichtung benötigter Komponenten wie dem Audiostream des Funks (mehr dazu in Kapitel 4.4)
- Einrichtung von Event-Listnern für die Datenbankmanipulation
- Einrichtung der Programmschleife. Um das Audiosignal kontinuierlich zu analysieren muss die entsprechende Funktion (updateData) regelmäßig (im Falle von ZVEI-Decoder 100 mal pro Sekunde) aufgerufen werden.

```
async mounted() {
  const audioStream = await getUserAudio()
  const { sfx, analyser } = await createContext(audioStream)
  this.context = sfx
  this.analyser = analyser

  ipcRenderer.send( channel: 'getEm', args: {} )

  ipcRenderer.on( channel: 'postEm', listener: (event, em) => {
    this.$store.dispatch( type: 'updateEinsatzmittel', em )
  })

  setInterval( handler: () => {
    this.updateData()
  }, this.timeoutTime)
},
```

4.3 Funktion updateData

updateData() ist die Hauptfunktion des Programms. In ihr werden alle Daten der Helferfunktionen gesammelt, weitergeleitet und eventuelle Ergebnisse abgespeichert.

Der grobe Ablauf der Funktion ist dabei wie folgt:

1. Überprüfung der momentanen Frequenz des Audiostreams und ob diese eine festgelegte Mindestlautstärke überschreitet, ist dies nicht der Fall endet die Ausführung und wird 10ms später erneut gestartet.
2. Hat das Programm eine Frequenz erkannt wird diese einem Ton nach Abb. 2 (Seite 4) zugeordnet, sollte die Frequenz mit keinem Ton übereinstimmen endet die Ausführung hier
3. Der erfasste Ton wird zwischengespeichert und ein Timer gestartet. 600 Millisekunden werden weiter

```
updateData() {  
  const currentFrequency = getCurrentFrequencyFft(  
    this.context,  
    this.analyser  
  )  
  if (currentFrequency !== -1) {  
    this.$store.dispatch( type: 'updateLastTimeReceived', Date.now())  
    const currentTon = getTonNummer(currentFrequency)  
    if (currentTon !== -1) {  
      this.tonFolgeGesamt.push(currentTon)  
      this.zeitSeitLetztemTon = Date.now()  
      if (this.tonFolgeGesamt.length === 1) {  
        setTimeout( handler: () => {  
          console.log(this.tonFolgeGesamt)  
          let tonfolge = getValidatedTonfolge(  
            this.tonFolgeGesamt,  
            this.minTonCount,  
            this.maxTonCount  
          )  
          if (  
            tonfolge == null &&  
            Date.now() - this.zeitSeitLetztemTon <= 210  
          ) {  
            this.intervallCheck = setInterval( handler: () => {  
              tonfolge = getValidatedTonfolge(  
                this.tonFolgeGesamt,  
                this.minTonCount,  
                this.maxTonCount  
              )  
              if (tonfolge != null) {  
                console.log('Tonfolge durch IntervallCheck ermittelt:')  
                this.$store.dispatch( type: 'addAlarm', tonfolge)  
                console.log(this.$store.state.currentTonfolge)  
                clearInterval(this.intervallCheck)  
                this.tonFolgeGesamt = []  
              } else if (  
                tonfolge == null &&  
                Date.now() - this.zeitSeitLetztemTon > 210  
              ) {  
                clearInterval(this.intervallCheck)  
                this.tonFolgeGesamt = []  
              }  
            }, timeout: 140)  
            } else if (tonfolge != null) {  
              console.log('Tonfolge ermittelt:')  
              this.$store.dispatch( type: 'addAlarm', tonfolge)  
              this.tonFolgeGesamt = []  
            } else {  
              this.tonFolgeGesamt = []  
            }  
          }, timeout: 600)  
        }  
      }  
    }  
  }  
}
```

Abb. 8: Funktion updateData in der Datei App.vue

Töne gesammelt und im Anschluss an eine Helfermethode gesendet. Da die Funktion `updateData` alle 10ms einen Durchlauf macht, wird jeder Ton einer Tonfolge (Dauer 70ms) mehrfach erkannt werden, deshalb müssen die gesammelten Töne validiert und auf fünf Töne zusammengefasst werden. Dieses Konzept dient dazu eventuelle Fehlererkennung aussortieren zu können.

4. Wurde eine Tonfolge erfolgreich validiert, wird diese abgespeichert, damit sie vom User-Interface gerendert werden kann.

4.4 audio.js

audio.js ist die erste von zwei JavaScript-Dateien, die für die Auswertung der Fünf-tonfolgen verantwortlich ist. Sie exportiert verschiedene Funktionen, die von der Hauptfunktion `updateData` aufgerufen werden

4.4.1 Funktion `getUserAudio`

In Zeile 1-3 findet man die Funktion `getUserAudio()`.

Diese Funktion macht sich das native `MediaDevices`-Interface von JavaScript zunutze, um Zugriff auf den Audioeingang des Benutzers zu bekommen.

Im Anschluss gibt sie den erhaltenen Audiostream an die Aufrufende Funktion zurück, damit diese ihn in einer Variable abspeichern kann.

4.4.2 Funktion `createContext`

Die Funktion `createContext()` nutzt wiederum den Stream, den sie als Parameter übermittelt bekommt, und erschafft um ihn herum einen sogenannten `AudioContext`. `AudioContext` ist eine Schnittstelle der zuvor in Kapitel 3.2 erwähnten `Web-Audio-API`. Ein erstellter `AudioContext` bietet Möglichkeiten zur Manipulation und Analyse des eingegebenen Audiosignals. Da das Audiosignal nicht manipuliert, sondern analysiert werden soll, wird hier noch eine `AnalyserNode` erstellt, die die Analysefunktionen der `Web-Audio-API` bereitstellt

```
async function getUserAudio() {
    return await navigator.mediaDevices.getUserMedia( constraints: { audio: true })
}

async function createContext(stream) {
    const sfx = new AudioContext( contextOptions: { sampleRate: 6000 })
    const source = sfx.createMediaStreamSource(stream)
    const analyser = sfx.createAnalyser()
    analyser.minDecibels = -40
    analyser.maxDecibels = -10
    analyser.smoothingTimeConstant = 0.3
    source.connect(analyser)

    return { sfx, analyser }
}

function getCurrentFrequencyFft(sfx, analyser) {
    const nyquist = sfx.sampleRate / 2
    const frequencyPerBin = nyquist / analyser.frequencyBinCount
    const frequencyData = new Uint8Array(analyser.frequencyBinCount)
    analyser.getByteFrequencyData(frequencyData)
    let peak = 0
    frequencyData.forEach( callbackfn: (v :number , i :number ) => {
        if (v > frequencyData[peak]) {
            peak = i
        }
    })
    if (frequencyData[peak] > 80) {
        return peak * frequencyPerBin
    } else {
        return -1
    }
}

export { getUserAudio, createContext, getCurrentFrequencyFft }
```

Abb. 9: audio.js

4.4.3 Funktion `getCurrentFrequencyFft`

Die Funktion `getCurrentFrequencyFft` führt die Fast-Fourier-Transformation durch, hierfür bekommt sie den zuvor erstellten `AudioContext` und die dazugehörige `AnalyserNode` als Parameter übermittelt.

Nachdem die Fast-Fourier-Transformation das Signal in seine Einzelfrequenzen aufgesplittet hat, wird die Frequenz mit dem höchsten Amplitudenwert (Lautstärke) an die Funktion `updateData` zurückgegeben.

4.5 zvei.js

audio.js liefert uns die Frequenz des analysierten Audiosegments. Die erste Funktion in zvei.js heißt `getTonNummer` und arbeitet nun mit dieser Frequenz, um sie einer Tonziffer entsprechend Abb. 2 (Seite 4) zuzuweisen.

4.5.1 Funktion `getTonNummer`

Hierzu werden zuerst alle Ziffern (0-9 und R) in einem Array deklariert. Für eventuelle Funktionserweiterungen wurde auch die Frequenz für den Sirenenauslöseton (S) mit eingefügt.

Im Anschluss wird die übergebene Frequenz mit einer gewissen Toleranz mit den Frequenzen der einzelnen Ziffern verglichen. Fällt die überlieferte Frequenz auf eine Ziffer, wird diese zurückgegeben, ansonsten wird der Wert -1 zurückgegeben.

4.5.2 Funktion `getValidatedTonfolge`

Die Aufgabe von `getValidatedTonfolge` ist die Reduzierung und Validierung einer Tonsammlung (Siehe Kapitel 4.3).

Eine Tonsammlung (siehe Abb. 11) beinhaltet alle Töne, die `updateData` innerhalb von 600ms nach Erkennung des ersten Tones erkannt hat. Durch die Ausführung von `updateData` alle 10 Millisekunden wird jeder Ton öfters erkannt, diese gilt es nun zu zusammenzufassen.

Im absoluten Idealfall würde jeder Ton 7 mal erkannt werden (70ms Tondauer / 10ms Ausführung), dieser Idealfall tritt jedoch seltenst ein.

```
function getTonNummer(f) {
  const frequenzen = [
    1060,
    1160,
    1270,
    1400,
    1530,
    1670,
    1830,
    2000,
    2200,
    2400,
    2600,
    1240
  ]
  const toleranz = 13
  let currentTon = -1
  frequenzen.forEach((value : number , index : number ) => {
    const min = value - toleranz
    const max = value + toleranz
    if (f >= min && f <= max) {
      if (index === 9) {
        currentTon = 0
      } else if (index === 10) {
        currentTon = 'R'
      } else if (index === 11) {
        currentTon = 'S'
      } else {
        currentTon = index + 1
      }
    }
  })
  return currentTon
}

function getValidatedTonfolge(tf, minTonCount, maxTonCount) {
  let counter = 0
  const result = []
  const counters = []
  tf.forEach((value, index) => {
    counter++
    if (value !== tf[index + 1] || index === tf.length - 1) {
      if (counter >= minTonCount && counter <= maxTonCount) {
        result.push(value)
        counters.push(counter)
      }
      counter = 0
    }
  })
  if (result.length > 5) {
    let lowestCount = 0
    while (result.length > 5) {
      counters.forEach((v, i : number ) => {
        if (v < counters[lowestCount]) {
          lowestCount = i
        }
      })
      console.log(lowestCount)
      result.splice(lowestCount, deleteCount: 1)
    }
  } else if (result.length < 5) {
    return null
  }
  result.forEach((value, index : number ) => {
    if (value === 'R') {
      result[index] = result[index - 1]
    }
  })
  return result
}

export { getTonNummer, getValidatedTonfolge }
```

Abb. 10: zvei.js

(37) [8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8] —ob
—: lt]

Abb. 11: Beispiel einer Tonsammlung, die an `getValidatedTonfolge` übergeben wird

Um jeden Ton genau sieben mal zu erkennen müsste die erste Ausführung von `updateData` genau auf den Anfang des ersten Tons fallen, ist dies nicht der Fall, wird `updateData` im Verlauf der Tonfolge auch genau zwischen zwei aufeinanderfolgenden Tönen ausgeführt, was einer Erkennung im Wege stehen kann. Hierdurch, und durch andere Einflüsse wie z.B. Störungen am Funk, kann es passieren, dass Töne öfter oder seltener als sieben mal erkannt werden.

Deshalb muss ein Bereich festgelegt werden, wie oft ein Ton in der Tonsammlung vorhanden sein darf, um von `getValidatedTonfolge` als Teil einer regelrechten Tonfolge erkannt zu werden. Dies geschieht durch die Parameter `minTonCount` und `maxTonCount`. Nach ausführlichen Tests unter unterschiedlichen Bedingungen wurden `minTonCount` auf 3 und `maxTonCount` auf 11 festgelegt. Mit diesen Werten wurde die bisher beste Balance aus zuverlässiger Erkennung und Reduzierung von Fehlerkennungen beobachtet. In späteren Versionen des Programms wird der Endnutzer diese Werte selbst an seine Gegebenheiten anpassen können.

Ist ein Ton in der Tonsammlung also mindestens so oft wie `minTonCount` und höchstens so oft wie `maxTonCount` vorhanden, gilt er als erkannt. Das Ergebnis hat dann im Idealfall genau fünf erkannte Töne, in diesem Fall gilt die gesamte Tonfolge als validiert und wird an `updateData` zurückgegeben.

[8, 9, 4, 5, 8]

Abb. 12: Die Tonsammlung aus Abb. 11 nach Durchlaufen von `getValidatedTonfolge`

Sind hingegen zu wenige Töne erkannt worden, gilt die Tonfolge als nicht validiert und wird verworfen, bei zu vielen erkannten Tönen wird die Tonfolge weiter reduziert, indem die Töne, die am nächsten an der Grenze des Bereiches zwischen `minTonCount` und `maxTonCount` gelöscht werden.

4.6 Die Einsatzmitteldatenbank

Damit erkannte Tonfolgen nicht ihrer rohen Form ausgegeben werden, sondern stattdessen die zugehörigen Einsatzmittel, kann der Nutzer den Tonfolgen einen Namen zuordnen (z.B. Kleine Schleife, große Schleife, o.Ä.). Diese Beziehung wird dann in einer lokalen Datenbank gespeichert und kann jederzeit geändert oder gelöscht werden.

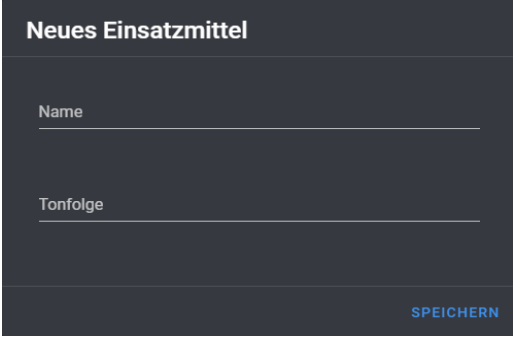
```
ipcMain.on( channel: 'getEm', listener: (evt : IpcMainEvent , searchObj) => {
  db.find(searchObj, projection: (err, docs) => {
    if (err) {
      throw err
    }
    console.log(docs)
    evt.reply('postEm', docs)
  })
})

ipcMain.on( channel: 'addEm', listener: (evt : IpcMainEvent , em) => {
  console.log('Inserting EM')
  db.insert(em, (err, newEm) => {
    console.log('Replying with: ' + newEm)
    evt.reply('emAdded', newEm)
  })
})

ipcMain.on( channel: 'editEm', listener: (evt : IpcMainEvent , em) => {
  console.log('Updating EM')
  db.update({ _id: em._id }, em)
})

ipcMain.on( channel: 'deleteEm', listener: (evt : IpcMainEvent , em) => {
  console.log('Deleting EM ' + em._id)
  db.remove({ _id: em._id })
})
```

Abb. 13: Datenbanklogik im Hintergrundprozess



The image shows a dark-themed dialog box with the title "Neues Einsatzmittel". Inside the dialog, there are two text input fields. The first field is labeled "Name" and the second field is labeled "Tonfolge". At the bottom right of the dialog, there is a button with the text "SPEICHERN" in all caps.

Abb. 14: Dialog zur Erstellung eines neuen Einsatzmittels

5 Anhang

5.1 Bedienungsanleitung

Die folgende Anleitung bezieht sich auf ZVEI-Decoder Version 0.2.6. Aufgrund fortlaufender Entwicklung kann sich das Programm mit zukünftigen Updates verändern.

5.1.1 Links

Quellcode: <https://github.com/david-breidert/zvei-decoder>

Download Version 0.2.6: <https://github.com/david-breidert/zvei-decoder/releases/tag/v0.2.6>

Download aktuellste Version: <https://github.com/david-breidert/zvei-decoder/releases>

5.1.2 Benötigte Hardware

- Ein PC mit Microsoft Windows
- ein Gerät zum Funkempfang mit Audioausgang (z.B. Ein Funkscanner) angeschlossen an den PC ODER ein Mikrofon sowie ein Fünftongerät oder eine aufgenommene Fünftonfolge

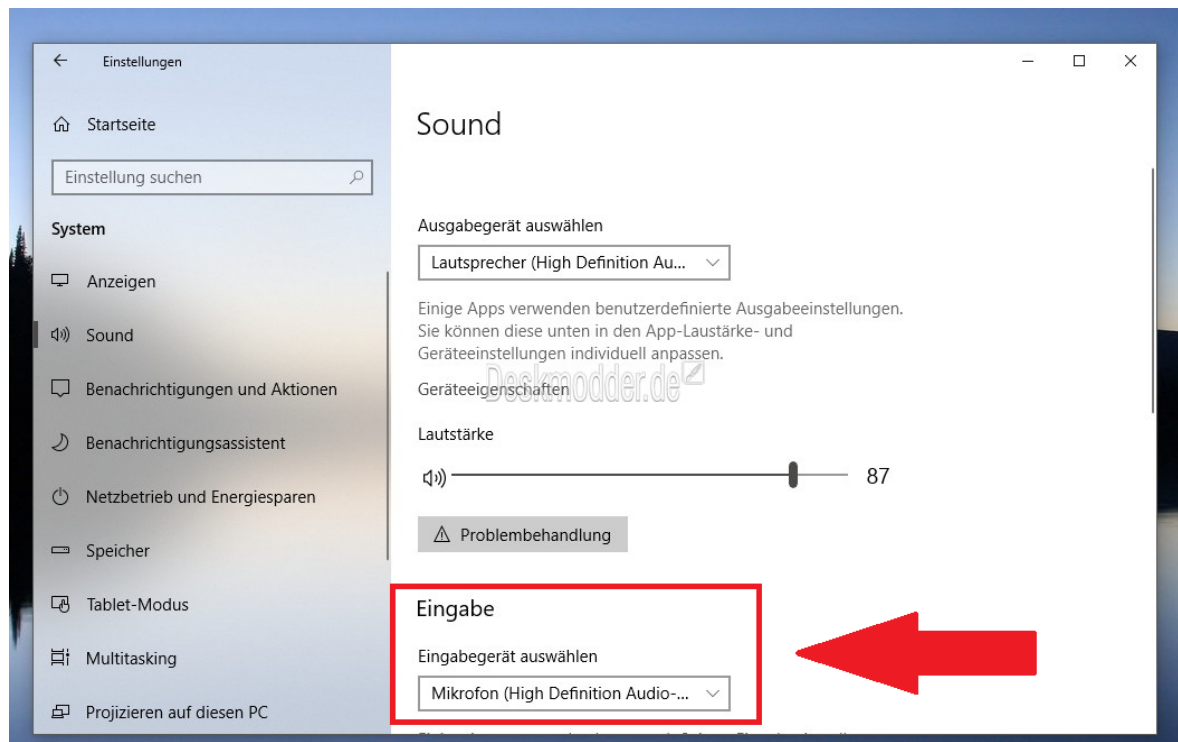
5.1.3 Installation

Nach Download und Ausführung des Installers (zvei-decoder-Setup-0.2.6.exe) wird das Programm installiert und nach der Installation automatisch geöffnet.

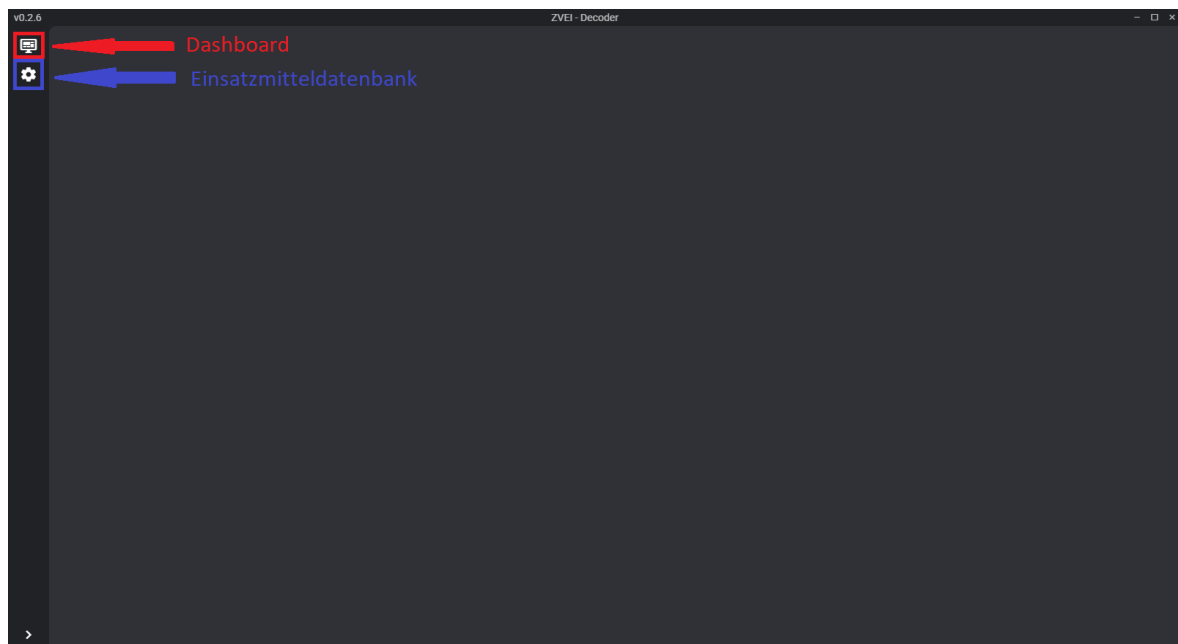
Sollte sich das Programm nicht automatisch öffnen, kann es über die erstellte Verknüpfung auf dem Desktop geöffnet werden.

5.1.4 Nötige Windows-Einstellungen

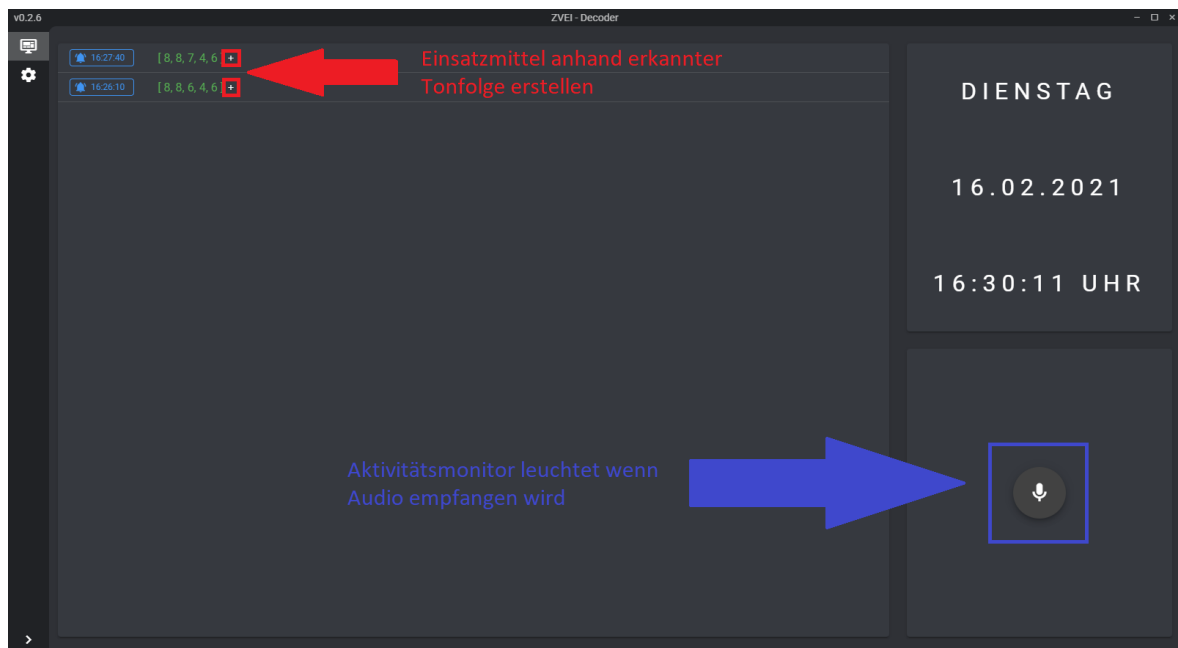
ZVEI-Decoder greift auf das Standard-Audioeingabegerät des Computers zu. Deshalb ist es wichtig, dass das Audiogerät, über das die Fünftonfolgen eingespielt werden in Windows als Standardgerät eingestellt ist.



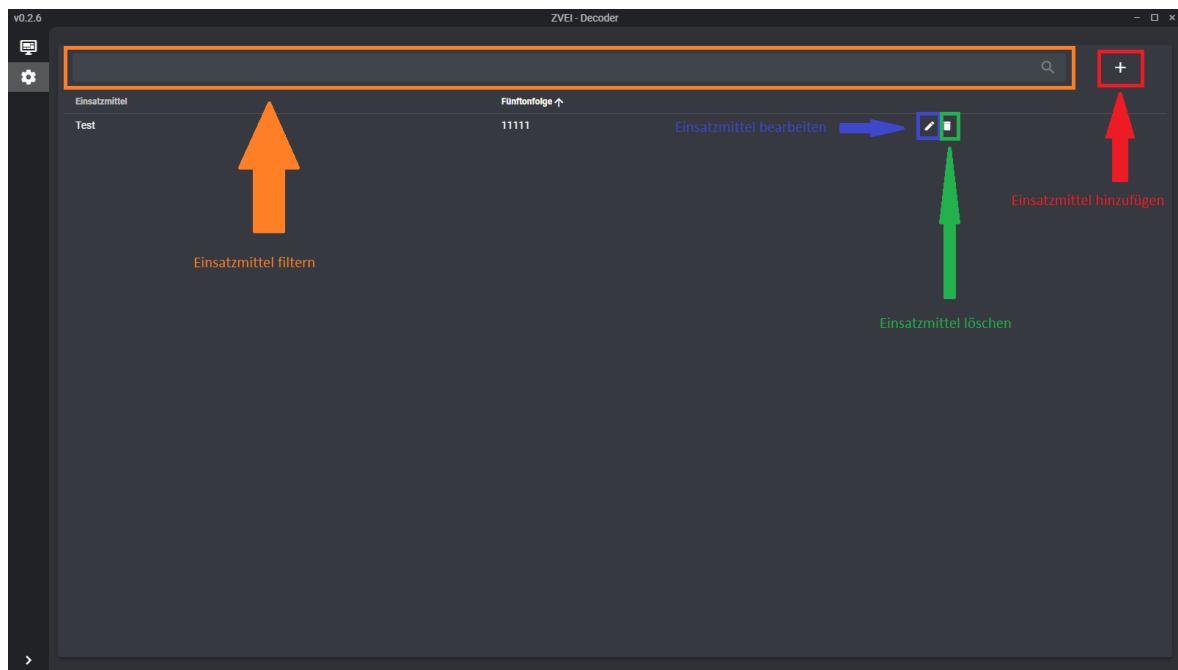
5.1.5 Nach dem Programmstart



5.1.6 Dashboard



5.1.7 Einsatzmitteldatenbank



[6][7][8][9][10]

Literaturverzeichnis

- 1: Thüringer Ministerium für Inneres und Kommunales, Brand- und Katastrophenschutzbericht 2018, 2018, https://innen.thueringen.de/fileadmin/Publikationen/Anlage_1_Jahresbericht_2018.pdf
- 2: Hochschule Karlsruhe, Klassen von Signalen: Kontinuierliche und diskrete Signale, , <https://www.eit.hs-karlsruhe.de/mesysto/teil-a-zeitkontinuierliche-signale-und-systeme/zeitkontinuierliche-signale/klassen-von-signalen/kontinuierliche-und-diskrete-signale.html>
- 3: W3C Audio Working Group, Web Audio APIW3C Candidate Recommendation Snapshot, 2021, <https://www.w3.org/TR/webaudio/>
- 4: Wikipedia, Autokorrelation, 2020, <https://de.wikipedia.org/wiki/Autokorrelation>
- 5: Wikipedia, Schnelle Fourier-Transformation, 2020, https://de.wikipedia.org/wiki/Schnelle_Fourier-Transformation
- 6: , Technische Richtlinie der Behörden und Organisationen mit Sicherheitsaufgaben (BOS): Geräte für die Funkalarmierung , 2000
- 7: Elias Oenal, multimon-ng, 2020, <https://github.com/EliasOenal/multimon-ng>
- 8: VueJS Team, VueJS API, 2021, <https://vuejs.org/v2/api/>
- 9: ElectronJS Team, , 2021, <https://www.electronjs.org/docs/api>
- 10: VuetifyJS Team, VuetifyJS Guide, 2021, <https://vuetifyjs.com/en/introduction/why-vuetify/>

Abbildungsverzeichnis

Abb. 1: typischer Funkmeldeempfänger (Quelle: https://www.gfd-katalog.com/ludwig_feuerschutz/quattro-xli/25597).....	2
Abb. 2: Tonfrequenzen, aus denen die Fünftönsfolgen aufgebaut werden (Quelle: https://de.wikipedia.org/wiki/5-Ton-Folge).....	4
Abb. 3: zeitlicher Ablauf einer Funkalarmierung Quelle: https://de.wikipedia.org/wiki/5-Ton-Folge	4

Abb. 4: Gegenüberstellung von analogen und digitalen Signalen (Quelle: https://upload.wikimedia.org/wikipedia/commons/f/f2/%C3%9Cbersicht_kontinuierliche_und_diskrete_Signale.svg).....	9
Abb. 5: vereinfachte graphische Darstellung einer Autokorrelation. Der blaue Graph wird so lange nach hinten verschoben, bis er über dem grünen liegt.....	10
Abb. 6: Dashboard-Ansicht von ZVEI-Decoder.....	11
Abb. 7: Datenbankansicht von ZVEI-Decoder mit geschwärzten Einsatzmitteln...	12
Abb. 8: Funktion updateData in der Datei App.vue.....	13
Abb. 9: audio.js.....	15
Abb. 10: zvei.js.....	17
Abb. 11: Beispiel einer Tonsammlung, die an getValidatedTonfolge übergeben wird.....	18
Abb. 12: Die Tonsammlung aus Abb. 11 nach Durchlaufen von getValidatedTonfolge.....	18
Abb. 13: Datenbanklogik im Hintergrundprozess.....	19
Abb. 14: Dialog zur Erstellung eines neuen Einsatzmittels.....	19

