Transmisja danych dźwiękiem w JavaScript od podstaw

Część 3: Własny stos sieciowy

W pierwszej części tej serii (Programista 8/2016) omówiliśmy szczegółowo zasadę działania prostego i intuicyjnego algorytmu Dyskretnej Transformaty Fouriera. W tej części pokażemy, jak ten sam algorytm przetestować na prawdziwych próbkach audio przy użyciu Web Audio API omówionego w części drugiej (Programista 1/2017). Oprócz tego przeanalizujemy metody modulacji cyfrowej oraz stworzymy krok po kroku prostą implementację stosu sieciowego bazującego na modelu OSI oraz TCP/IP. Umożliwi nam on zrealizowanie aplikacji przesyłającej wiadomości tekstowe między dwoma urządzeniami tylko za pomocą dźwięku.

Najważniejszym elementem naszego stosu sieciowego jest warstwa fizyczna. To w niej zachodzi cały proces Cyfrowego Przetwarzania Sygnałów. Prędkość transmisji będzie zależeć w dużej mierze od rozwiązań DSP, jakie w niej zastosujemy. Faktem jest, iż Web Audio API udostępnia wiele „gotowców", których można by zwyczajnie użyć bez wgłębiania się w szczegóły. Założeniem tej serii była jednak od początku chęć zrozumienia tych procesów od podstaw. Z tego powodu naszą przygodę z implementacją własnego stosu sieciowego zaczniemy od prostego i intuicyjnego algorytmu Dyskretnej Transformaty Fouriera, który omawialiśmy w części pierwszej. Teorię mamy już za sobą, więc w tej części skupimy się tylko na praktycznym zastosowaniu zdobytej wiedzy. Ułatwi to zrozumienie pochodzenia wyników zwracanych przez elementy wbudowane w Web Audio API takie jak `AnalyserNode`. Finalnie całe DSP zamkniemy w klasie `PhysicalLayer`. Będzie ona pełnić rolę „modemu", na którym opierać się będą wyższe warstwy naszego stosu sieciowego. Przy ich implementacji będziemy się wzorować na istniejących już rozwiązaniach, takich jak ramka Ethernet czy protokół TCP.

Tworząc nasz system, będziemy starać się dokładać nowe elementy krok po kroku na zasadzie ewolucji, a nie rewolucji. Założeniem jest stworzenie rozwiązania prostego, który składa się tylko z elementów niezbędnych do poprawnej pracy. Co ważne, nie skorzystamy z żadnych bibliotek zewnętrznych. Do dyspozycji będzie zatem przeglądarka, czysty JavaScript oraz Web Audio API. Myślę, że dzięki takiemu podejściu wszystkie elementy układanki staną się bardziej zrozumiałe.

WAVEANALYSER ORAZ WAVEGENERATOR, CZYLI DSP PO NASZEMU

Implementacja algorytmu Dyskretnej Transformaty Fouriera omówiona w pierwszej części tego artykułu opierała się na luźnych funkcjach. W tej części posłużymy się tym samym algorytmem, jednak w nieco bardziej uporządkowanej formie, jaką jest klasa `WaveAnalyser`. Do kompletu, oprócz `WaveAnalyser`, przygotowano także klasę `WaveGenerator` umożliwiającą generowanie próbek fali o podanych parametrach.

Klasa `WaveAnalyser` ma dwie zależności – `Buffer` oraz `Complex`. Pierwsza to prosta implementacja bufora cyklicznego, druga to podstawowe operacje na liczbach zespolonych. Podsumowując, zrefaktoryzowany kod zagadnień z części pierwszej znajdziemy w poniższych plikach:

- https://audio-network.rypula.pl/buffer-class
- https://audio-network.rypula.pl/complex-class
- https://audio-network.rypula.pl/wave-analyser-class
- https://audio-network.rypula.pl/wave-generator-class

Przykład użycia naszych klas zaprezentowano w Listingu 1:

Listing 1. Przykład użycia klas WaveAnalyser oraz WaveGenerator

```
var
  SAMPLE_PER_PERIOD = 32,
  WINDOW_SIZE = 1024,
  APPLY_WINDOW_FUNCTION = false,
  UNIT_PHASE = 0.25,
  AMPLITUDE = 0.7,
  wg = new WaveGenerator(SAMPLE_PER_PERIOD),
  wa = new WaveAnalyser(
    SAMPLE_PER_PERIOD,
    WINDOW_SIZE,
    APPLY_WINDOW_FUNCTION
  ),
  sampleList = [],
  fb,
  i;

// generowanie tablicy próbek
wg.setUnitPhase(UNIT_PHASE);
wg.setAmplitude(AMPLITUDE);
for (i = 0; i < WINDOW_SIZE; i++) {
  sampleList.push(wg.getSample());
  wg.nextSample();
}

// odtworzenie parametrów fali z próbek
for (i = 0; i < WINDOW_SIZE; i++) {
  wa.handle(sampleList[i]);
}
fb = wa.getFrequencyBin();
```

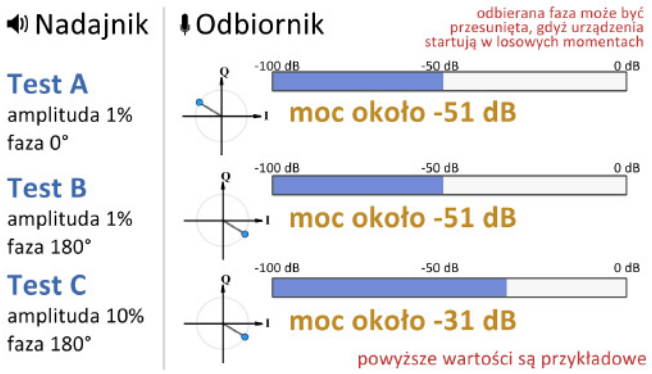
```
cLog(fb.getReal().toFixed(2)); // -358.40
cLog(fb.getImaginary().toFixed(2)); // -0.00
cLog(wa.getUnitPhase().toFixed(2)); // 0.25
cLog(wa.getAmplitude().toFixed(2)); // 0.70
```

WAVEANALYSER ORAZ WAVEGENERATOR – TESTY PRAKTYCZNE

Ok, nadszedł czas, by przetestować nasze własne rozwiązania DSP w praktyce. W tym celu wykorzystamy klasę `AudioMonoIO`, którą zaimplementowaliśmy w części drugiej tej serii. Opakowuje ona Web Audio API w nieco prostszy interfejs.

Przykładowa aplikacja umożliwia zarówno nadawanie, jak i odbieranie dźwięku. Parametry fali możemy zmieniać za pomocą widgetów. Mając do dyspozycji dwa urządzenia, możemy poeksperymentować z generowaniem częstotliwości oraz strojeniem się na nią na drugim urządzeniu. Jeżeli dysponujemy tylko jednym urządzeniem, możemy skorzystać z trybu „loopback". Linki do przykładu znajdziemy poniżej:

» https://audio-network.rypula.pl/wave-generator-wave-analyser
» https://audio-network.rypula.pl/wave-generator-wave-analyser-src



Rysunek 1. Testy praktyczne naszych klas WaveAnalyser oraz WaveGenerator

Prześledźmy zatem przykładowe scenariusze testowe wykorzystujące dwa urządzenia (Rysunek 1). Urządzenie A nadaje stały ton o częstotliwości 1500 Hz. Urządzenie B nasłuchuje na tej samej częstotliwości, pokazując przy tym moc odbieranej fali w formie poziomego paska. Gdy wartość amplitudy nadawanej fali zmieni się dziesięciokrotnie, powoduje to zmianę odbieranej mocy o około 20 dB. Dodatkowo obok paska możemy odczytać przesunięcie fazowe odbieranej fali w formie wskazówki. Urządzenia startują z losowym przesunięciem, więc nie oczekujmy, że faza odbieranej fali będzie zgodna co do wartości. Łatwiej zobaczymy różnice tych zmian. Na przykład gdy w nadajniku dodamy do fazy fali 180 stopni, spowoduje to w odbiorniku nagły obrót wskazówki o 180 stopni. Przypomnijmy: informacja o przesunięciu fazowym jest dla nas niedostępna z poziomu „gotowców" Web Audio API. Węzeł `AnalyserNode` zwraca niestety tylko wartości bezwzględne liczb zespolonych otrzymanych z wyjścia FFT.

PROBLEM SYNCHRONIZACJI CZĘSTOTLIWOŚCI A DRYFOWANIE FAZY

Eksperymentowanie z poprzednim przykładem pozwala zaobserwować dość ciekawy efekt nieco niezgodny z intuicją. Otóż gdy ustawimy identyczną częstotliwość zarówno na odbiorniku, jak i nadajniku, nasza wskazówka fazy w teorii powinna stać w miejscu. Tak się jednak nie dzieje. Czasem kręci się wolniej, czasem szybciej, czasem w lewo, a czasem w prawo. Jest to spowodowane błędem odwzorowania częstotliwości w układach audio naszych urządzeń. Oscylatory są wrażliwe na zmiany temperatury. Między innymi to powoduje brak synchronizacji fali odbieranej z falą lokalnie wygenerowaną w obiekcie klasy `WaveAnalyser`. Nawet drobny błąd stanie się wyraźnie widoczny w formie dryfowania fazy (Rysunek 2).

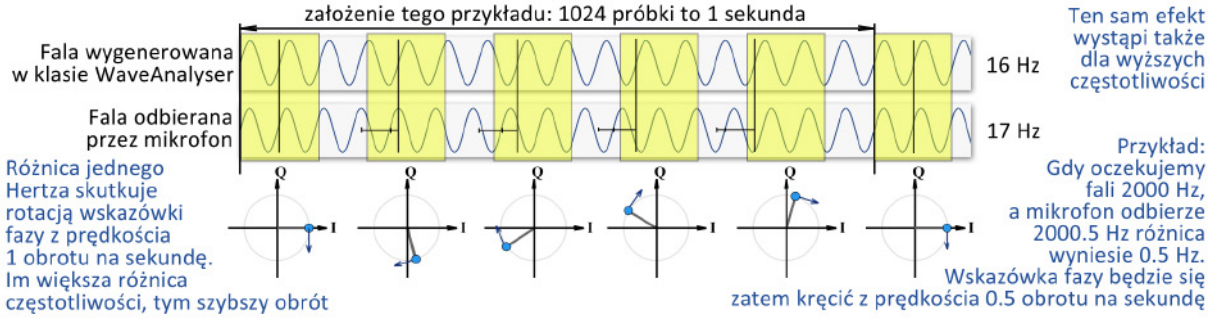
Różnice w częstotliwościach wprowadza także efekt Dopplera. Jeżeli zbliżamy się do odbiornika, wtedy częstotliwość wydaje się być wyższa, gdy oddalamy – niższa. W efekcie zmianie ulega prędkość i kierunek obrotu naszej wskazówki. Ten efekt możemy łatwo wykorzystać do stworzenia aplikacji sprawdzającej, czy odchodzimy od komputera, czy do niego wracamy ;)

W naszej przykładowej aplikacji problem ten możemy zniwelować poprzez zmianę częstotliwości za pomocą widgetów. Gdy urządzenia nie przemieszczają się względem siebie, na ogół mówimy o korektach rzędu setnych części Hz. Niestety nawet tak drobny błąd spowoduje obrót wskazówki o znaczny kąt po upływie kilku sekund. Widać zatem wyraźnie, że metody modulacji wykorzystujące fazę fali nośnej są nieco trudniejsze w realizacji, gdyż wymagają dokładnego dostrojenia odbiornika.

Opisywany wyżej efekt nie wystąpi jednak w trybie „loopback". Wtedy sample krążą w obrębie jednego urządzenia.

WAVEANALYSER KONTRA ANALYSERNODE

W tej sekcji pokażemy, jak użyć naszej klasy `WaveAnalyser` do wygenerowania wykresu widma amplitudowego. W pierwszej części tego artykułu już to robiliśmy, jednak jedynie dla sygnałów syntetycznych. Wtedy też, dla uproszczenia, na osi poziomej używaliśmy wartości wyrażonych w `samplePerPeriod`. Zrobiliśmy tak tylko dlatego, by nasze klasy nie musiały przyjmować dwóch parametrów (`frequency` oraz `sampleRate`), lecz tylko jeden (`samplePerPeriod`). Fakt użycia na osi stałych odstępów wartości `samplePerPeriod` powodował nienaturalne rozciągnięcie wy-



Rysunek 2. Efekt dryfowania fazy związany z brakiem synchronizacji częstotliwości

kresu, przez co nasze 3 sinusoidy nie ukazywały się jako „piki" o tej samej szerokości. W większości przypadków na wykresach widma na osi poziomej używa się liniowej skali wyrażonych w Hertzach. W naszym przypadku, aby to skorygować, jedyne, co musimy zrobić, to zamienić Hertze na `samplePerPeriod` (Listing 2).

Listing 2. Użycie WaveAnalyser do wygenerowania wykresu widma amplitudowego

```
function getFrequencyData(timeDomainData) {
  var
    samplePerPeriod = 1, // zmienimy i tak w pętli
    windowSize = timeDomainData.length,
    frequencyBinCount = 0.5 * windowSize,
    windowFunction = true,
    waveAnalyser = new WaveAnalyser(
      samplePerPeriod,
      windowSize,
      windowFunction
    ),
    N = timeDomainData.length,
    frequencyData = [],
    decibel,
    i,
    k;

  // wypełnienie próbkami
  for (i = 0; i < timeDomainData.length; i++) {
  waveAnalyser.handle(timeDomainData[i]);
  }

  // zmiana samplePerPeriod nie kasuje
  // próbek już znajdujących się w oknie
  for (k = 0; k < frequencyBinCount; k++) {
    samplePerPeriod = (k === 0)
      ? Infinity       // DC-offset (0 Hz)
      : N / k;
    waveAnalyser.setSamplePerPeriod(
      samplePerPeriod
    );
    decibel = waveAnalyser.getDecibel();
    frequencyData.push(decibel);
  }

  return frequencyData;
}
```

`AnalyserNode` zwraca parę wyników z dziedziny czasu oraz częstotliwości (`getFrequencyData`, `getTimeDomainData`). Jeżeli przepuścimy dane z dziedziny czasu przez naszą klasę i porównamy otrzymane wyniki z dziedziną częstotliwości otrzymaną z `AnalyserNode`, powinniśmy otrzymać te same rezultaty. Po przedstawieniu danych graficznie widzimy, że oba widma wyglądają tak samo (Rysunek 3). Udało nam się zatem własnoręcznie zaimplementować odpowiednik metody `getFrequencyData` z `AnalyserNode`.

Rysunek 3. Nasza klasa WaveAnalyser kontra AnalyserNode

Testy możemy wykonać także samodzielnie, korzystając z przykładu:

» https://audio-network.rypula.pl/wave-analyser-vs-analyser-node
» https://audio-network.rypula.pl/wave-analyser-vs-analyser-node-src

Teraz może się nam nasuwać pytanie – po co w ogóle pisać własne klasy DSP, skoro mamy Web Audio API, w którym ktoś to zrobił za nas? W sumie racja, jednak od początku założeniem tej serii było nie tylko pokazanie, jak używać „gotowców" takich jak `AnalyserNode`, ale także omówienie podstaw Cyfrowego Przetwarzania Sygnałów. Unikamy dzięki temu „magicznego pudełka", które robi coś, o czym nie mamy zielonego pojęcia. Oczywiście Szybka Transformata Fouriera (FFT) użyta w `AnalyserNode` jest nieporównywalnie wydajniejsza, za to nasza metoda jest jednak relatywnie prostsza do zrozumienia. Dodatkowo nasza klasa `WaveAnalyser` zwraca fazę fali oraz umożliwia dostrojenie się na konkretną częstotliwość. W FFT częstotliwości prążków widmowych są mniej elastyczne.

WYDAJNOŚĆ WAVEANALYSER

Zastanówmy się teraz, jaką wydajność oferuje nasza klasa. By odpowiedzieć na to pytanie, posłużymy się poniższym przykładem:

» https://audio-network.rypula.pl/wave-analyser-performance
» https://audio-network.rypula.pl/wave-analyser-performance-src

Wyniki testu przeprowadzone na komórce oraz laptopie przedstawiono na Rysunku 4.

Laptop

Rozmiar okna	Czas trwania okna	Średni czas przetwarzania jednego prążka	Szacunkowa liczba prążków real-time	Szacunkowy czas przetwarzania DFT
1024	21.3 ms	0.8 ms (3.8% czasu okna)	max 27	0.410 s
2048	42.7 ms	0.54 ms (1.3% czasu okna)	max 79	0.553 s
4096	85.3 ms	1.06 ms (1.2% czasu okna)	max 81	2.171 s
8192	170.7 ms	2.13 ms (1.2% czasu okna)	max 80	8.724 s
16384	341.3 ms	4.35 ms (1.3% czasu okna)	max 78	35.635 s
32768	682.7 ms	8.67 ms (1.3% czasu okna)	max 79	142.049 s
65536	1365.3 ms	17.7 ms (1.3% czasu okna)	max 77	579.994 s

Liczba prążków DFT to połowa rozmiaru okna

Komórka

Rozmiar okna	Czas trwania okna	Średni czas przetwarzania jednego prążka	Szacunkowa liczba prążków real-time	Szacunkowy czas przetwarzania DFT
1024	21.3 ms	2.03 ms (9.5% czasu okna)	max 11	1.039 s
2048	42.7 ms	1.35 ms (3.2% czasu okna)	max 32	1.382 s
4096	85.3 ms	1.97 ms (2.3% czasu okna)	max 43	4.035 s
8192	170.7 ms	3.97 ms (2.3% czasu okna)	max 43	16.261 s
16384	341.3 ms	7.83 ms (2.3% czasu okna)	max 44	64.143 s
32768	682.7 ms	15.69 ms (2.3% czasu okna)	max 44	257.065 s
65536	1365.3 ms	31.61 ms (2.3% czasu okna)	max 43	1035.796 s

Liczba prążków DFT to połowa rozmiaru okna

Rysunek 4. Wydajność naszej klasy WaveAnalyser

W tabelach przedstawiono średnie czasy potrzebne do przetworzenia jednej częstotliwości dla różnych rozmiarów okna. Rozmiary te są potęgami dwójki tak jak ma to miejsce w `AnalyserNode`. Z wyników możemy także odczytać, jakim procentem czasu trwania okna jest czas potrzebny na przetworzenie jednej częstotliwości. Przekroczenie 100% oznaczałoby, że urządzenie nie jest w stanie przetwarzać sampli w czasie rzeczywistym. Dzieląc czas trwania okna przez czas przetwarzania jednej częstotliwości, możemy zgrubnie oszacować maksymalną liczbę częstotliwości, jakie jest w stanie przetwarzać nasze urządzenie w czasie rzeczywistym. Dla średniej klasy laptopa ta liczba wynosi około 80, dla komórki około 40. Test na komputerze stacjonarnym z procesorem taktowanym częstotliwością 3.6 GHz dał wynik około 150 prążków. Ostatnią kolumną jest szacunkowy czas przetwarzania wyniku identycznego do tego, jaki otrzymalibyśmy z FFT. Nasz prosty algorytm ma złożoność $O(N^2)$, więc czasy te rosną bardzo szybko.

Obciążenie procesora w 100% nie byłoby jednak dobrym pomysłem, gdyż nasza strona zwyczajnie przestałaby reagować. Możemy jednak ten problem obejść, przenosząc obliczenia do osobnego wątku (Web Worker).

Oczywiście wydajność będzie zależeć od konkretnego urządzenia, lecz nawet taka zgrubna analiza daje nam pewien ogólny obraz o możliwościach naszej klasy `WaveAnalyser`.

PRZETWARZANIE OFFLINE

Jeśli wydajność naszych klas okaże się dla nas niewystarczająca, ciągle pozostaje możliwość przetwarzania sygnałów w trybie „of-

fline". Mam tu na myśli nagranie dźwięku do dużej tablicy próbek i przeanalizowanie go później. Podobnie możemy postąpić na urządzeniu nadającym. Wystarczy wcześniej przygotować tablicę sampli, by na końcu przesłać je na głośniki przy użyciu węzła `AudioBufferSourceNode`. Akurat tej klasy nie opisaliśmy w części 2 tej serii, dlatego chętnych odsyłam do dokumentacji Web Audio API:

» https://developer.mozilla.org/en-US/docs/Web/API/AudioBufferSourceNode

Dodatkowo polecam poniższy przykład:

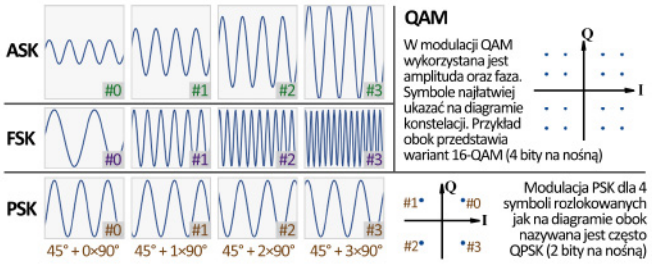
» https://audio-network.rypula.pl/wav-file-recorder
» https://audio-network.rypula.pl/wav-file-recorder-src

DSP zrealizowane „offline" pozwoli zapomnieć o ograniczeniach wydajności szczególnie urządzeń mobilnych. Jeżeli urządzenie jest wolne, możemy zwyczajnie rozciągnąć cały proces w czasie.

METODY MODULACJI CYFROWEJ

Każdą sinusoidę możemy opisać za pomocą trzech parametrów – amplitudy, częstotliwości oraz fazy. Jak zapewne podpowiada nam intuicja, odpowiednie sterowanie (modulacja) tymi trzema wartościami umożliwia przesyłanie danych.

Na początku musimy zdecydować o liczbie unikalnych wartości, które zmieniać będą falę nośną. Na przykład chcąc przesłać 3 bity w konkretnej jednostce czasu, należałoby ustalić 8 symboli (2^3), które znać będzie zarówno nadajnik, jak i odbiornik. Prześledźmy zatem, jakie metody modulacji mamy do dyspozycji.



Rysunek 5. Podstawowe metody modulacji cyfrowej

ASK – kluczowanie amplitudy (Amplitude-Shift Keying)

W tej metodzie kolejne symbole zmieniają tylko amplitudę fali nośnej (Rysunek 5). Chcąc przesłać 4 symbole, moglibyśmy zmapować je na przykład na następujące wartości amplitudy (głośności): symbol 0 – 25%, symbol 1 – 50%, symbol 2 – 75%, symbol 3 – 100%. Symbole możemy oczywiście rozlokować inaczej, na przykład zacząć od 0%, czyli całkowicie wytłumionej nośnej.

Niestety ta metoda modulacji fali nośnej jest dość podatna na zakłócenia. Na przykład, gdyby podczas transmisji danych ktoś na chwilę przesłonił mikrofon lub zwyczajnie przeszedł między dwoma urządzeniami, zmieniłaby się moc odbieranej fali w odbiorniku. Mogłoby to spowodować odebranie zupełnie innego symbolu niż oczekiwany. Efekt byłby tym silniejszy, im więcej byłoby unikalnych symboli. Pewnym obejściem problemu jest użycie tylko dwóch symboli (B-ASK, Binary ASK). Wtedy nasza transmisja oczywiście „rozciągnęłaby" się w czasie. Przesłanie bajta przy użyciu B-ASK trwałoby 8x dłużej niż w przypadku skorzystania z 256 unikalnych poziomów amplitudy.

Odbiornik możemy zrealizować przy użyciu `AnalyserNode` lub naszej klasy `WaveAnalyser`. W obydwu przypadkach jesteśmy w stanie odczytać moc odbieranej fali wyrażoną w decybelach. Przyjmijmy, że amplituda symbolu „zero" jest 10x mniejsza od symbolu „jeden". Przekłada się to na różnicę wynoszącą 20 dB między naszymi dwoma symbolami. Na przykład, gdy dla symbolu „zero" otrzymamy wartość -42 dB, to możemy oczekiwać, że dla symbolu „jeden" otrzymamy wartość około -22 dB. Gdyby nasze „zero" miało 100x mniejszą amplitudę niż „jeden", wtedy różnica wyniosłaby 40 dB. Dla 1000x mniejszej amplitudy różnica wyniosłaby 60 dB itd.

FSK – kluczowanie częstotliwości (Frequency-Shift Keying)

W tej metodzie amplituda oraz faza fali jest stała. Nasze symbole modulują więc częstotliwość (Rysunek 5). Nadajnik możemy łatwo zrealizować za pomocą `OscillatorNode` lub `WaveGenerator`. Działanie odbiornika sprowadza się do badania głośności wszystkich częstotliwości, na jakich mogą wystąpić symbole. Znalezienie maksimum w ustalonym wcześniej paśmie da nam w rezultacie nadawany symbol. Nie jest to problem zarówno dla `AnalyserNode`, jak i naszej klasy `WaveAnalyser`.

Jak zapewne podpowiada nam intuicja, ta metoda jest znacznie lepsza od ASK. To prawda. Dodatkowo jest dużo lepiej skalowalna. Dołożenie kolejnych symboli sprowadza się do użycia szerszego zakresu częstotliwości. Oczywiście tutaj też w pewnym momencie napotkamy limit, jednak nie jest on tak odczuwalny. Na zwykłym sprzęcie audio jesteśmy w stanie bezproblemowo uzyskać ponad 300 symboli. W przypadku ASK byłoby to bardzo ciężkie w realizacji.

PSK – kluczowanie fazy (Phase-Shift Keying)

W tej metodzie to amplituda oraz częstotliwość jest stała. Jedyne, co się zmienia, to faza fali nośnej (Rysunek 5). Dla człowieka transmisja tego typu jest odbierana jako seria takich samych „pików". Zwyczajnie na drodze ewolucji nie wykształciła się w nas zdolność bezpośredniego słyszenia przesunięcia fazowego.

Ta metoda wiąże się z pewną trudnością. Wymagane jest bardzo dokładne dostrojenie odbiornika do odbieranego sygnału. To, dlaczego tak się dzieje, omówiliśmy nieco wcześniej przy okazji praktycznych testów `WaveAnalyser`. Dodatkowo metoda ta jest wrażliwa na wszelkie „potknięcia" w wydajności naszego urządzenia. Jeżeli zgubimy kilka sampli, nasz dalszy strumień może być już nieco przesunięty. Może to powodować odbieranie błędnych symboli. Pamiętajmy, że każda sekunda zawiera ponad 40 tysięcy sampli, a okres fali jest rzędu kilkudziesięciu sampli (`samplePerPeriod` to `sampleRate/frequency`). Jeżeli „po drodze" zgubimy kilka próbek, skutkować to będzie przesunięciem. Będzie ono wyraźnie widoczne w przeskokach fazy.

Na Rysunku 5 przedstawiono wygląd fali dla różnych symboli w dziedzinie czasu oraz diagram konstelacji, na którym najłatwiej zobrazować symbole przesyłane metodami PSK czy QAM. Diagramu konstelacji używaliśmy także w części pierwszej tego artykułu.

QAM – kwadraturowa modulacja amplitudowo-fazowa (Quadrature Amplitude Modulation)

Ta metoda to w dużym skrócie połączenie modulacji amplitudy oraz fazy. Umożliwia zwiększenie liczby symboli możliwych do przesłania w jednostce czasu (Rysunek 5).

TTS Company

Microsoft Partner
Gold Volume Licensing

TTS Company rekomenduje oprogramowanie Microsoft ®

www.OprogramowanieKomputerowe.pl

Microsoft Azure Office 365 OneDrive

Więcej informacji: (22) 272 94 94 sales@tts.com.pl

Sprzedaż ▸▸ **Dystrybucja** ▸▸ **Import na zamówienie**

TTS Company Sp. z o.o. ul.Domaniewska 44A 02-672 Warszawa e-mail: tts@tts.com.pl www.tts.com.pl

OFDM – ORTOGONALNE ZWIELOKROTNIANIE W DZIEDZINIE CZĘSTOTLIWOŚCI (ORTHOGONAL FREQUENCY-DIVISION MULTIPLEXING)

Chcąc uzyskać dużo większe prędkości transmisji danych, możemy skorzystać z techniki OFDM. Polega ona na przesyłaniu danych na wielu częstotliwościach nośnych w tym samym czasie. Każda z tych nośnych może być modulowana dowolną omówioną wcześniej metodą. Tego typu rozwiązanie działa np. w technologii LTE, Wi-Fi, DVB-T (naziemna telewizja cyfrowa) i wielu innych.

Co oznacza słowo „ortogonalne" w nazwie metody? Bez wgłębiania się w szczegóły, chodzi o to, że gdy umiejętnie dobierzemy częstotliwości nośnych, nie będą one się wzajemnie zakłócać. Dobór ten polega na zapewnieniu, że w czasie trwania symbolu OFDM liczba cykli każdej z fal składowych jest liczbą całkowitą. Dzięki temu nie wystąpi efekt wycieku widma, o którym pisaliśmy w części 1. Użycie funkcji okna staje się zbędne, gdyż energia każdej z częstotliwości składowych trafi tylko i wyłącznie do swojego prążka widmowego. Możemy więc umieścić składowe w sąsiadujących ze sobą prążkach bez obawy o wzajemne zakłócanie. Rozwiniemy nieco to zagadnienie przy okazji omawiania rozkładu energii w prążkach. Technika OFDM pozwala więc na gęste upakowanie transmisji w wąskim zakresie częstotliwości, co oszczędza dostępne spektrum.

Do generowania symbolu OFDM idealnie nadaje się metoda `setPeriodicWave` z klasy `AudioMonoIO`, którą zaimplementowaliśmy i szczegółowo opisaliśmy w poprzedniej części artykułu. Wystarczy tylko wytłumić amplitudę tonu podstawowego, a transmitować jedynie dalsze harmoniczne przy dodatkowym modulowaniu ich amplitudy (ASK), fazy (FSK) lub obydwu tych parametrów (QAM). Przykład z 4 podnośnymi przedstawiono w Listingu 3. Jego wizualizację dla `sampleRate` 44.1 kHz przedstawiono na Rysunku 6. Użycie większego rozmiaru okna wydłuża czas jego trwania. Skutkuje to zmniejszeniem liczby symboli w ciągu sekundy. To jednak pozwala na upakowanie jeszcze większej liczby podnośnych w tym samym wycinku spektrum.

Listing 3. Generowanie symbolu OFDM przy użyciu naszej klasy AudioMonoIO

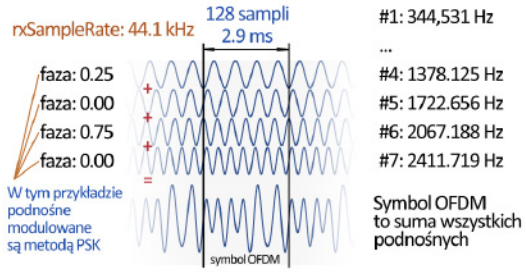
```
var
  SAMPLE_PER_OFDM_SYMBOL = 128,
  am = new AudioMonoIO(),
  baseFreq = am.getSampleRate() /
    SAMPLE_PER_OFDM_SYMBOL;

am.setPeriodicWave(
  baseFreq, // ton podstawowy
  1,        // głośność bazowa
  0,        // globalne przesunięcie fazy
  /* numer harmonicznej:
   1  2  3    4    5    6    7  ... */
  [0, 0, 0, 1.00, 1.0, 1.00, 1.0],// amplituda
  [0, 0, 0, 0.25, 0.0, 0.75, 0.0] // faza
  /*        ^^^^  ^^^  ^^^^  ^^^
  podnośne:  #1    #2   #3    #4    */
);
```

Link do klasy `AudioMonoIO` znajdziemy poniżej:

» https://audio-network.rypula.pl/audio-mono-io-class

Ten sam efekt możemy także osiągnąć poprzez zsumowanie wyników pracy kilku instancji naszej klasy `WaveGenerator`.

Jak widzimy, metoda OFDM pozwala znacznie przyspieszyć prędkość przesyłania danych. Nie ma jednak róży bez kolców. Jedną z trudności jest kwestia synchronizacji początku symbolu OFDM. Innym problemem jest kompensacja efektu Dopplera związanego z ruchem urządzenia. Te i inne trudności wymuszają stosowanie np. tonów pilotażowych o znanej częstotliwości i fazie. Jest to jednak szeroki temat, dlatego dociekliwych odsyłam do źródeł zewnętrznych.



Rysunek 6. Symbol OFDM pod lupą

TESTOWANIE METOD MODULACJI

Do eksperymentowania z generowaniem sampli dla różnych metod modulacji możemy posłużyć się przykładem „Modulation types". Dodatkowo umożliwia on wizualne zweryfikowanie 2 sekund próbek audio z dziedziny czasu. Jeżeli zobaczymy, że nasza nagrana fala jest nieciągła lub pojawiły się „dziury" o wartościach zerowych, oznacza to, że nasze urządzenie „nie wyrabia". Będzie to powodować problemy szczególnie przy technikach takich jak PSK, QAM czy OFDM. Rozwiązaniem może być zwiększenie wartości `bufferSize` bądź zmiana przeglądarki na inną. Linki do przykładu poniżej:

» https://audio-network.rypula.pl/modulation-types
» https://audio-network.rypula.pl/modulation-types-src

JAKĄ METODĘ MODULACJI WYBRAĆ?

Na tym etapie warto zastanowić się, jaka metoda modulacji cyfrowej będzie dla nas najlepsza. Dodatkowo musimy pamiętać, że duża część użytkowników korzystać będzie z urządzeń mobilnych. Należy zatem znaleźć kompromis pomiędzy wydajnością, prostotą implementacji, niezawodnością działania oraz prędkością transmisji danych.

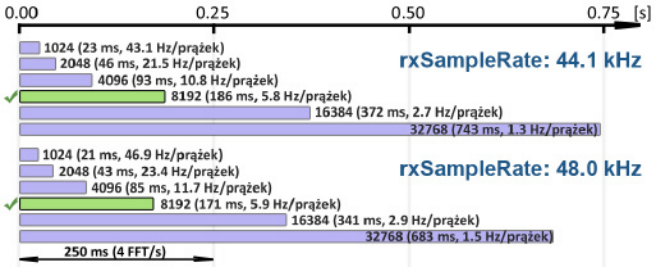
Na samym początku możemy odrzucić metodę ASK, gdyż jest nieodporna na zakłócenia. Techniki wykorzystujące fazę fali są kuszące, jednak ich realizacja jest dość trudna. To samo możemy powiedzieć o OFDM. W myśl zasady KISS (*ang. Keep It Simple, Stupid*) lepszym wyborem byłoby stworzenie rozwiązania prostego bez zbędnych komplikacji. Zostaje nam więc metoda FSK. Może być ona z powodzeniem zrealizowana przy użyciu naszych klas, jednak ich wydajność ograniczyłaby maksymalną liczbę unikalnych symboli. Biorąc pod uwagę wyniki naszych testów dla urządzeń mobilnych oraz zakładając pewien bufor bezpieczeństwa, moglibyśmy uzyskać maksymalnie kilkanaście symboli. Jest to jednak trochę za mało, by przesyłać dane z akceptowalną prędkością. Co zatem zrobić?

Na szczęście ze wszystkich wspomnianych do tej pory metod modulacji FSK jest dość łatwa do zaimplementowania na tym, co Web Audio API daje nam „za darmo". Mam tu na myśli `AnalyserNode` oraz `OscillatorNode`. Przypomnijmy: węzeł `AnalyserNode` „pod maską" używa algorytmu Szybkiej Transformaty Fouriera (FFT). Dla ścisłości: działanie `OscillatorNode` jest podobne. Zamienia on parametry opisujące harmoniczne na dziedzinę czasu, wykonując odwrotną FFT. W obu przypadkach obliczenia są często wspierane sprzętowo. Dodatkowo wszystko działa na

osobnym wątku przeglądarki, więc sumaryczny zysk wydajnościowy jest naprawdę duży. W dalszej części tego artykułu pójdziemy zatem tą drogą, zostawiając sobie furtkę na przyszłość umożliwiającą rozbudowę systemu o technikę OFDM.

ANALYSERNODE – SZYBKOŚĆ DOSTARCZANIA NOWYCH TABLIC FFT

Działanie algorytmu FFT polega na zamianie sampli dziedziny czasu na prążki widmowe dziedziny częstotliwości. Im więcej sampli trafi do okna FFT (`fftSize`), tym większa będzie rozdzielczość w dziedzinie częstotliwości. Wiąże się to jednak ze zwiększeniem czasu trwania okna. To z kolei wydłuży czas trwania naszego symbolu, wpływając negatywnie na opóźnienia (Rysunek 7).



Rysunek 7. Analiza różnych konfiguracji AnalyserNode

To, jakie parametry wybierzemy, zależy jednak głównie od prędkości dostarczania nowych tablic wyników FFT. Testy w tym zakresie możemy przeprowadzić przy użyciu poniższego przykładu:

» https://audio-network.rypula.pl/analyser-node-performance
» https://audio-network.rypula.pl/analyser-node-performance-src

Test opiera się na fakcie, że metodę `getFloatFrequencyData` z klasy `AnalyserNode` możemy wywoływać wielokrotnie, nawet gdy nowy wynik FFT nie jest jeszcze gotowy. W takim przypadku zwyczajnie dostaniemy tablicę o takich samych wartościach jak w poprzednim wywołaniu. Nasz skrypt porównuje więc aktualną tablicę wyników z poprzednią – jeśli coś się zmieniło, oznacza to, że dysponujemy nowym wynikiem FFT. Analizując czasy, możemy obliczyć wydajność `AnalyserNode` na danym urządzeniu.

Na Rysunku 8 przedstawiono wyniki testu dla różnych urządzeń. Dość wyraźnie widać, że urządzenia mobilne, takie jak komórki, są kilka razy wolniejsze od np. przeciętnego laptopa. Testy na szybkim sprzęcie stacjonarnym dały wyniki zbliżone do tych z laptopa. Nie ma natomiast większej różnicy między samą wielkością `fftSize` z uwagi na jego złożoność $O(N*\log_2(N))$. Dodatkowo sam algorytm działa na osobnym wątku, więc nie ma tutaj problemu z przycinaniem strony.

Laptop

Rozmiar okna (FFT size)	Czas trwania okna	Średni czas obliczania FFT	Najgorszy czas obliczania FFT
2048	42.7 ms	10.03 ms (99.69 FFT/s)	13.00 ms (76.92 FFT/s)
8192	170.7 ms	9.90 ms (101.02 FFT/s)	13.80 ms (72.46 FFT/s)
32768	682.7 ms	9.98 ms (100.18 FFT/s)	15.40 ms (64.94 FFT/s)

Komórka

Rozmiar okna (FFT size)	Czas trwania okna	Średni czas obliczania FFT	Najgorszy czas obliczania FFT
2048	42.7 ms	38.53 ms (25.95 FFT/s)	118.8 ms (8.42 FFT/s)
8192	170.7 ms	58.35 ms (17.14 FFT/s)	120.8 ms (8.28 FFT/s)
32768	682.7 ms	62.06 ms (16.11 FFT/s)	133.0 ms (7.52 FFT/s)

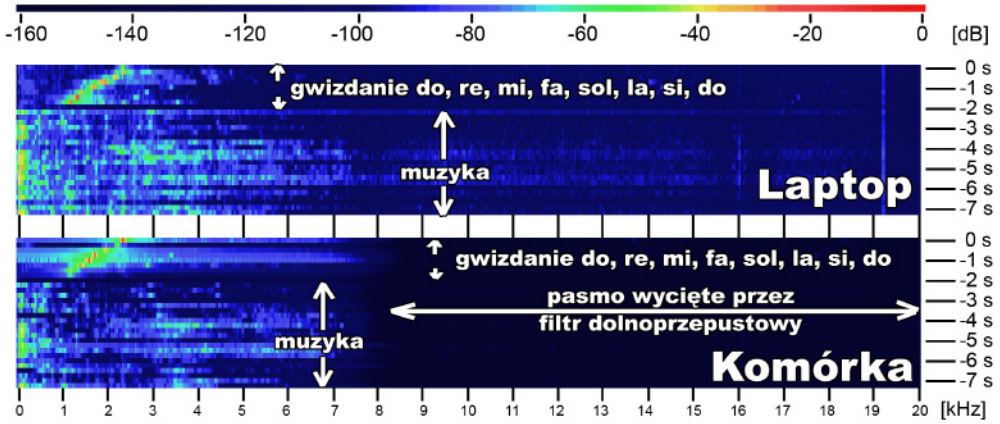
Rysunek 8. Szybkość dostarczania nowych wyników FFT przez AnalyserNode

Przypomnijmy: naszym celem jest stworzenie systemu transmisji danych, który działałby zarówno na laptopie, jak i komórce. Oznacza to, że wybierając częstotliwość próbkowania tablic wyników FFT, musimy wziąć pod uwagę najgorsze możliwe wyniki z urządzeń mobilnych. Testy wykazały, że najniższy wynik, jaki otrzymaliśmy, to około 7 FFT/s. Oznacza to, że co około 150 ms `AnalyserNode` jest w stanie w zasadzie w 100% zwrócić nam nową tablicę wyników FFT. Dla zachowania dodatkowego marginesu bezpieczeństwa my użyjemy jeszcze niższej wartości 4 FFT/s (odstępy co 250 ms). Rozmiar okna FFT nie może być zatem większy niż 250 ms, by nie „zachodzić" na zakładkę na następny blok sampli. Z Rysunku 7 możemy odczytać, że najlepszym wyborem będzie dla nas okno o rozmiarze `fftSize` równym 8192, gdyż trwa ono w granicach 186 ms/171 ms. Takiej właśnie wartości będziemy używać nieco dalej przy implementacji warstwy fizycznej.

SPECTRAL WATERFALL

W metodzie FSK symbole rozlokowane są na różnych częstotliwościach. Decyzja o tym, jaki symbol aktualnie odbieramy, sprowadza się do znalezienia prążka o największej wartości mocy w ustalonym wcześniej zakresie. Z kolei poziom szumu możemy obliczyć jako średnią wartość pozostałych prążków z zakresu.

W tej części zbadamy, jakie pasmo będzie najlepsze do realizacji naszego pomysłu. Od tego zależeć będzie liczba symboli FSK, a co za tym idzie – prędkość transmisji. W tym celu posłużymy się spektrogramem. Pozwala on analizować, jak widmo amplitudowe zmienia się w czasie. Wartości mocy wyrażone w decybelach otrzy-



Rysunek 9. Spektrogram

mują różne kolory, tak by można było łatwo ocenić, która część widma zawiera więcej energii, a która mniej. Z pomocą przychodzi klasa `Spectrogram`, której kod znajdziemy poniżej:

» https://audio-network.rypula.pl/spectrogram-class

Na Rysunku 9 przedstawiono 7 sekund takiej wizualizacji dla zakresu częstotliwości od 0 Hz do 20 kHz.

Analiza wyników z różnych urządzeń pozwala stwierdzić, iż na większości przeglądarek mobilnych możemy spotkać się z „dziwnym" efektem w okolicach 6.5 – 7.5 kHz. Powyżej tego zakresu moc odbieranych fal gwałtownie spada. Jest to związane z tym, że sygnał z mikrofonu przepuszczany jest przez filtr dolnoprzepustowy. Testy na Androidzie pokazały, że przeglądarki mobilne często korzystają ze strumienia audio, który jest używany do rozmów telefonicznych. Można to łatwo zweryfikować poprzez próbę zmiany głośności na karcie przeglądarki korzystającej z mikrofonu. Zwyczajnie zamiast paska o nazwie „Media" zobaczymy pasek „Głośność podczas rozmowy". Niestety nie mamy wpływu na implementację przeglądarki, więc najbezpieczniej jest założyć, że nasze pasmo jest ograniczone „od góry" częstotliwością 6 kHz. Po prostu nie możemy sobie pozwolić na to, by niektóre urządzenia odfiltrowały część naszej transmisji.

Jeżeli chodzi o ograniczenie „z dołu", sprawa wygląda różnie w zależności od tego, czy dźwięk jest odbierany, czy nadawany. Z odbiorem niskich częstotliwości w zasadzie nie powinno być problemu. Praktycznie każdy mikrofon jest w stanie odbierać dźwięki o częstotliwościach od 50 Hz. Gorzej jest z nadawaniem, gdyż wbudowane w laptopy czy komórki głośniki nie radzą sobie dobrze z niskimi częstotliwościami. Testy wykazały, że problemy zaczynają się pojawiać około 350 Hz. Sygnały o częstotliwościach mniejszych niż 150 Hz były już w zasadzie niemożliwe do wygenerowania. Dopiero lepszej klasy zestaw z subwooferem pozwalał zejść znacznie niżej.

Dla zachowania pewnego marginesu bezpieczeństwa możemy założyć, że „bezpieczny" zakres częstotliwości, jakim dysponujemy, leży w przedziale od 500 Hz do 6 kHz. Daje to szerokość pasma równą 5.5 kHz. Dla przyjętego przez nas `fftSize` równego 8192, w tym zakresie częstotliwości mieści się około 1000 prążków widmowych ((6000-500)/(sampleRate/8192)).

Linki do przykładów umożliwiających testy we własnym zakresie znajdziemy poniżej:

» https://audio-network.rypula.pl/spectral-waterfall
» https://audio-network.rypula.pl/spectral-waterfall-src
» https://audio-network.rypula.pl/spectral-waterfall-recorder
» https://audio-network.rypula.pl/spectral-waterfall-recorder-src

44.1 KHZ CZY 48 KHZ? OTO JEST PYTANIE

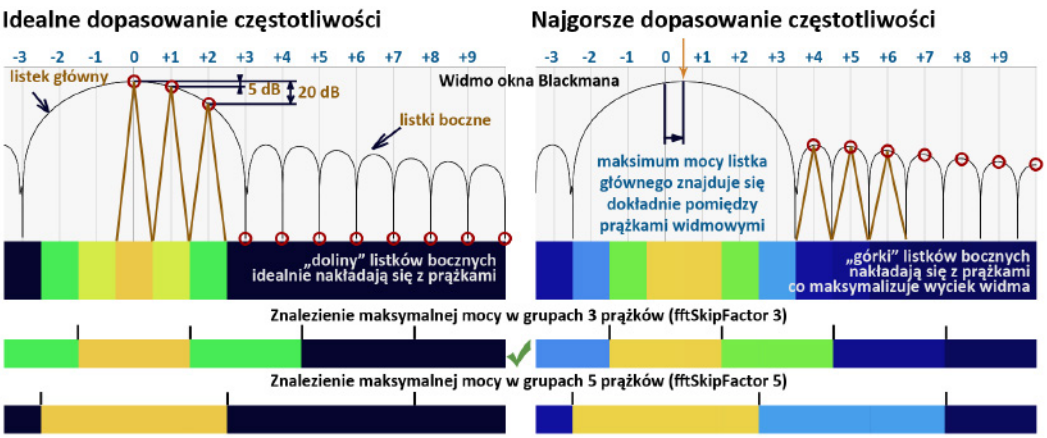
Przykład „Spectral Waterfall" umożliwia dodatkowo generowanie prostych tonów o częstotliwościach odpowiadających dokładnie prążkom widmowym po stronie odbiornika. Taka synchronizacja częstotliwości maksymalizuje moc sygnału. Niestety w przypadku `AnalyserNode` to nadajnik musi nastroić się na odbiornik. Jest to trochę dziwne, gdyż zazwyczaj jest odwrotnie. Trudno sobie wyobrazić, że dzwonimy do stacji radiowej i prosimy, żeby to nadajnik przestroił się na nasz odbiornik FM. W naszym przypadku nie stanowi to jednak problemu, gdyż łączymy tylko dwa urządzenia. W dodatku są one oddalone maksymalnie o kilka metrów, więc łatwo sprawdzić bądź zmienić ich parametry.

Powodem całego zamieszania jest fakt, iż częstotliwości prążków, oprócz wartości `fftSize`, zależą także od parametru `sampleRate`. Ten z kolei jest specyficzny dla urządzenia i nie jest możliwe użycie innej wartości. Z tego powodu w sekcji „Transmit" musimy podać częstotliwość próbkowania odbiornika. W Web Audio API w 99% przypadków będą to wartości równe 44100 Hz lub 48000 Hz.

ROZKŁAD ENERGII W PRĄŻKACH WIDMOWYCH

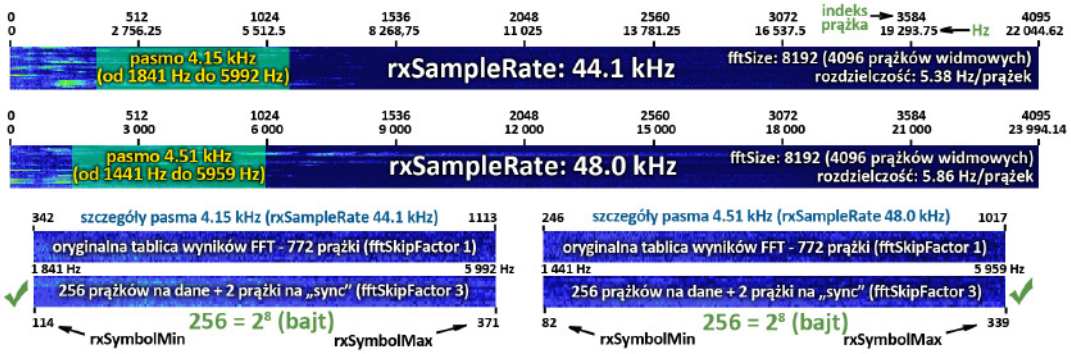
Pierwsza myśl podpowiada nam, że gdy nadawana częstotliwość będzie idealnie dopasowana do częstotliwości prążka, wtedy na spektrogramie zobaczymy silny sygnał dokładnie w jednej kolumnie. Tak jednak się nie dzieje. Energia fali zamiast trafić tylko do jednego prążka, jest dodatkowo rozproszona na kilka sąsiednich. Tryb „loopback" pozwala zbadać ten efekt dla warunków idealnych. Wtedy sample z wyjścia trafiają bezpośrednio na wejście w ramach tego samego urządzenia bez zakłóceń.

Efekt „rozproszenia" jest wynikiem zastosowania funkcji okna. Pod maską `AnalyserNode` działa okno Blackmana. Jego wygląd i porównanie z naszym spektrogramem przedstawiono na Rysunku 10. Niestety w Web Audio API nie przewidziano możliwości zmiany funkcji okna lub jego wyłączenia. Warto jednak wiedzieć, że w świecie DSP okno Blackmana jest jednym z wielu, jakie są dostępne. Na przykład w części pierwszej tej serii użyliśmy okna o nazwie Blackman–Nuttall. Różni się ono nieco sposobem dystrybucji energii naszej fali na wykresie widma. Generalnie im bardziej chcielibyśmy obniżyć poziom prążków niezwiązanych z falą, tym szerszy/grubszy będzie obszar bliski naszej fali. To jednak zmniejsza zdolność rozdzielczą fal o bardzo zbliżonej częstotliwości. Jak



Rysunek 10. Wygląd widma okna Blackmana na tle prążków widmowych


sages
MINIKONFERENCJA
DATA SCIENCE
I BIG DATA
W ZARZĄDZANIU
15 GRUDNIA 2017
REJESTRACJA BEZPŁATNA
EKSPERCI
DOMINIK BATORSKI
ANNA WRÓBLEWSKA
MARCIN CHOIŃSKI
JAKUB NOWACKI
LOKALIZACJA
AKADEMIA
LEONA KOŹMIŃSKIEGO
W WARSZAWIE
UL. JAGIELLOŃSKA 57/59
WWW.SAGES.COM.PL/DATA-SCIENCE-W-ZARZADZANIU



Rysunek 11. Wybór najlepszego pasma dla naszej warstwy fizycznej

widać, wybór okna zależy od konkretnego zastosowania. Zainteresowanych odsyłam do przeszukania Internetu dla frazy „fft window function comparison".

Podobny obrazek do tych znalezionych w Internecie możemy wygenerować sami, korzystając z poniższego przykładu:

» https://audio-network.rypula.pl/window-function
» https://audio-network.rypula.pl/window-function-src

W przypadku `AnalyserNode` i użytego w nim okna Blackmana widmo fali o idealnie dopasowanej częstotliwości rozciąga się wizualnie na 5 relatywnie „silnych" prążków (Rysunek 10). Tak naprawdę wartości prążków są punktem na krzywej widma okna. Główną, centralną część nazywamy listkiem głównym (ang. *main lobe*). Dalsze z nich noszą nazwę listków bocznych (ang. *side lobe*). Analiza prążków wchodzących w skład naszych listków pozwala stwierdzić, że różnica mocy między prążkiem głównym a jego dwoma najbliższymi sąsiadami wynosi tylko około 5 dB. Różnica między dalszymi sąsiadami wzrasta do 20 dB. Cała reszta prążków nakłada się idealnie na „doliny" widma okna. Doliny mają już tak niską wartość energii, że brakuje nawet koloru do jej pokazania.

Niestety nieco gorzej jest w przypadku braku dopasowania częstotliwości. W najgorszym przypadku każdy ze „szczytów" listków bocznych wypada idealnie w miejscu prążka widmowego. Wtedy wyciek widma jest największy, tworząc długi ślad daleko poza maksimum mocy. Ślad ten jest jednak i tak wielokrotnie słabszy niż gdybyśmy okna nie zastosowali w ogóle.

W ramach dygresji warto wspomnieć, że gdy fala jest idealnie dopasowana do danego prążka, nie jest w ogóle konieczne stosowanie funkcji okna. Brak okna możemy traktować jako okno prostokątne (mnożenie sampli dziedziny czasu przez same jedynki). „Szczyt" listka głównego wpływa wtedy na wartość mocy tylko jednego „centralnego" prążka. Wszystkie inne prążki nakładają się idealnie na „doliny" widma okna. Sprawia to, że wartość ich mocy jest pomijalnie mała. Korzyści z takiego stanu rzeczy czerpie technika OFDM. Niestety nawet niewielki brak dopasowania częstotliwości może spowodować katastrofalny wyciek widma, gdyż listki boczne widma okna prostokątnego są bardzo „wysokie". Widać wyraźnie, że chcąc rezygnować z funkcji okna, musimy dokładnie wiedzieć, co robimy. Można więc odnieść wrażenie, że twórcy `AnalyserNode` celowo nas przed tym ustrzegli, zwyczajnie blokując możliwość wyłączenia odgórnie ustalonego okna Blackmana ;)

By nieco „stuningować" prążki zwracane przez `AnalyserNode`, możemy zwyczajnie połączyć prążki w grupy, wybierając największą wartość mocy w każdej z nich. Przy idealnym dopasowaniu częstotliwości połączenie w grupy po 5 prążków dałoby praktycznie idealny wykres z tylko jedną kolumną o dużej mocy. By oszczędzić dostępne pasmo, poprzestaniemy jednak na połączeniu tylko 3 prążków w jeden. Zapewni to odstęp około 20 dB pomiędzy mocą prążka sygnału a następnego z kolei prążka o dużej mocy wynikającego z czystej matematyki. Jeżeli wykryjemy różnicę mniejszą niż 20 dB, oznaczać to będzie, że inny sygnał będący *de facto* szumem niebezpiecznie zbliża się do poziomu naszego prawdziwego sygnału.

W przykładzie „Spectral Waterfall" łączeniem prążków możemy sterować za pomocą kontrolki „FFT skip factor".

MAPOWANIE PRĄŻKÓW WIDMOWYCH NA SYMBOLE FSK

Biorąc pod uwagę to, co przeanalizowaliśmy do tej pory, możemy finalnie ustalić rozlokowanie oraz liczbę symboli FSK. Większość urządzeń, jakie chcemy obsługiwać, jest zdolne odbierać dźwięki w zakresie częstotliwości od 500 Hz do 6 kHz. Dla `fftSize` równego 8192 mieści się w tym przedziale około 1000 prążków. Chcąc zapewnić wyraźny odstęp mocy prążka sygnału od poziomu szumu, zdecydowaliśmy się grupować prążki po 3. Redukuje to liczbę prążków w naszym paśmie do około 333. Jak się okazuje, jest to ciągle bardzo przyzwoita liczba, gdyż przesłanie jednego bajta „naraz" wymaga 256 symboli. Nasze bezpieczne pasmo będzie zatem wykorzystane tylko częściowo.

Do naszych obliczeń założymy jednak 258 symboli. Dwa dodatkowe symbole użyjemy do synchronizacji, o czym będzie nieco dalej. Pamiętajmy też, że nie możemy przekroczyć 6 kHz z uwagi na filtr dolnoprzepustowy występujący głównie na urządzeniach mobilnych. Chcąc jak najmniej pokrywać się z dźwiękami życia codziennego (np. gwizdanie to zakres od 1 do 2 kHz), nasze finalne pasma dosuniemy maksymalnie do 6 kHz.

Ostatecznie dla próbkowania 44.1 kHz otrzymamy zakres od około 1841 kHz do 5992 kHz, a dla próbkowania 48 kHz zakres od około 1441 Hz do 5959 kHz (Rysunek 11). Koniec zakresu dla 44.1 kHz jest nieco przesunięty względem 48 kHz, tak aby symbole „sync" nie zachodziły na siebie. O synchronizacji będziemy jednak mówić nieco dalej.

EFEKT DOPPLERA

Gdy nasze urządzenie nadawcze zbliża się bądź oddala od odbiornika, wpływa to na wartość odbieranej częstotliwości. Na wykresie widma nasz prążek może się zatem przesunąć w lewo bądź w prawo. Nominalna rozdzielczość widma amplitudowego dla `fftSize` równego 8192 to około 5.4 Hz (44100/8192) lub 5.9 Hz (48000/8192). Połączenie 3 prążków w jeden zwiększa te wartości trzykrotnie. Otrzymamy więc wartości kolejno 16.1 Hz oraz 17.6 Hz.

Spacerujący człowiek porusza się z prędkością około 1 m/s, co w najgorszym przypadku przekłada się na różnicę w odbieranej częstotliwości rzędu 4.4 Hz dla fali 1500 Hz (1500*343/(343±1)) lub nawet 17-18 Hz dla fali 6000 Hz (6000*343/(343±1)). Do obliczeń przyjęto prędkość dźwięku równą 343 m/s. Dla dolnej części naszego pasma błąd wynikający z efektu Dopplera nie będzie jeszcze wpływał negatywnie na odbierany symbol. Niestety, im wyższy numer prążka, tym efekt Dopplera jest bardziej widoczny. Pamiętajmy więc o tym, spacerując po pokoju podczas transmisji. Istnieje duże ryzyko, że spowodujemy tym odbieranie błędnych symboli.

PRÓBKOWANIE TABLIC WYNIKÓW FFT

Zastanówmy się teraz, jaką prędkość transmisji możemy uzyskać dla ustalonych do tej pory parametrów. Przyjęliśmy, że nawet najwolniejsze urządzenie, jakie chcemy obsługiwać, jest zdolne do przetwarzania 4 tablic FFT na sekundę (każda co 250 ms). Z każdej z tablic jesteśmy w stanie odczytać jeden bajt, gdyż dla danych mamy do dyspozycji 256 unikalnych symboli. Z pozoru może się więc wydawać, że możemy osiągnąć prędkość rzędu 4 bajtów na sekundę, czyli 32 bps. Niestety w rzeczywistości ta liczba będzie dwukrotnie mniejsza z uwagi na częstotliwość Nyquista, o której mówiliśmy w poprzedniej części artykułu. Ostatecznie więc nadajnik musi pracować z częstotliwością 2 symboli na sekundę, co przekłada się na prędkość transmisji równą 2 bajty na sekundę 16 bps (Rysunek 12).

Odstępy rzędu 250 ms są już na tyle duże, że nie musimy ich obliczać z dokładnością do pojedynczego sampla. Możemy je z powodzeniem zrealizować za pomocą tradycyjnego JavaScriptowego `setInterval`. Zapewnia on, że nasze odstępy nie będą „dryfowały" w czasie. Nawet gdy jeden handler się spóźni, czasy wszystkich następnych nie ulegną przesunięciu – 0.0, 0.5, 1.2 (spóźniony handler), 1.5, 2.0, 2.5 itd. Oprócz `setInterval` możemy także użyć `setTimeout`, jednak wtedy musielibyśmy sami korygować opóźnienia, posiłkując się czasem systemowym. Takie rozwiązanie implementuje poniższa klasa:

» https://audio-network.rypula.pl/smart-timer-class

PROBLEM SYNCHRONIZACJI

Częstotliwość próbkowania będąca dwukrotnością częstotliwości nadawania symboli skutkuje tym, że poprawne wartości leżą albo w tablicach FFT o numerach parzystych (0, 2, 4, 6, ... – `rxSampleOffset` 0), albo nieparzystych (1, 3, 5, 7, ... – `rxSampleOffset` 1). Zależy to od tego, z jakim przesunięciem czasowym wystartowały nasze urządzenia. Idealny moment to ten, w którym nasze okno FFT obejmuje tylko jeden symbol. Najgorszy moment to ten, w którym wyłapujemy symbol zarówno aktualny, jak i poprzedni. Skąd zatem wiadomo, które przesunięcie powinniśmy wybrać? Pomoże nam w tym transmiter, poprzez przesyłanie specjalnego wzorca, który wypatrywać będzie odbiornik. Wzorzec o najlepszych parametrach pozwoli podjąć decyzję o wyborze przesunięcia.

Do detekcji wzorca posłużymy się korelacją. Zwraca ona wartość liczbową mówiącą o tym, jak bardzo nasza sekwencja jest do niego podobna (Listing 4).

Listing 4 . Wyznaczanie wartości współczynnika korelacji

```
var
  syncSeq = [1, -1, 1, -1],
  a = [-1, -1, -1, 0],
  b = [-1, 1, -1, 1],
  c = [1, -1, 1, -1];

function getCorrV(code, v) {
  var i, result = 0;

  for (i = 0; i < code.length; i++) {
    result += code[i] * v[i];
  }

  return result;
}

console.log(getCorrV(synSeq, a)); // -1
console.log(getCorrV(synSeq, b)); // -4
console.log(getCorrV(synSeq, c)); // 4
```

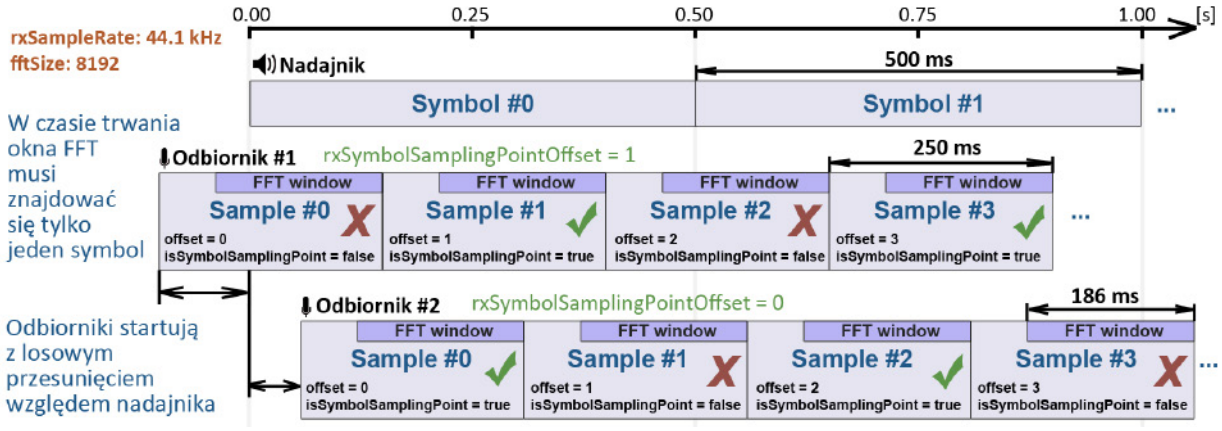
W przykładzie z Listingu 4 za nasz wzorzec przyjęliśmy sekwencję czterech naprzemiennych -1 i 1. Dla ciągu „a" nasza korelacja jest bliska zeru. Oznacza to brak podobieństwa. Dla sekwencji „b" oraz „c" otrzymamy kolejno -4 i 4. Długość wzorca wynosi 4. Oznacza to, że w dwóch ostatnich przypadkach odnaleźliśmy maksimum podobieństwa. Różnica polega tylko na odbiciu znaków sekwencji. Kod klasy do obliczania korelacji możemy znaleźć poniżej:

» https://audio-network.rypula.pl/correlator-class

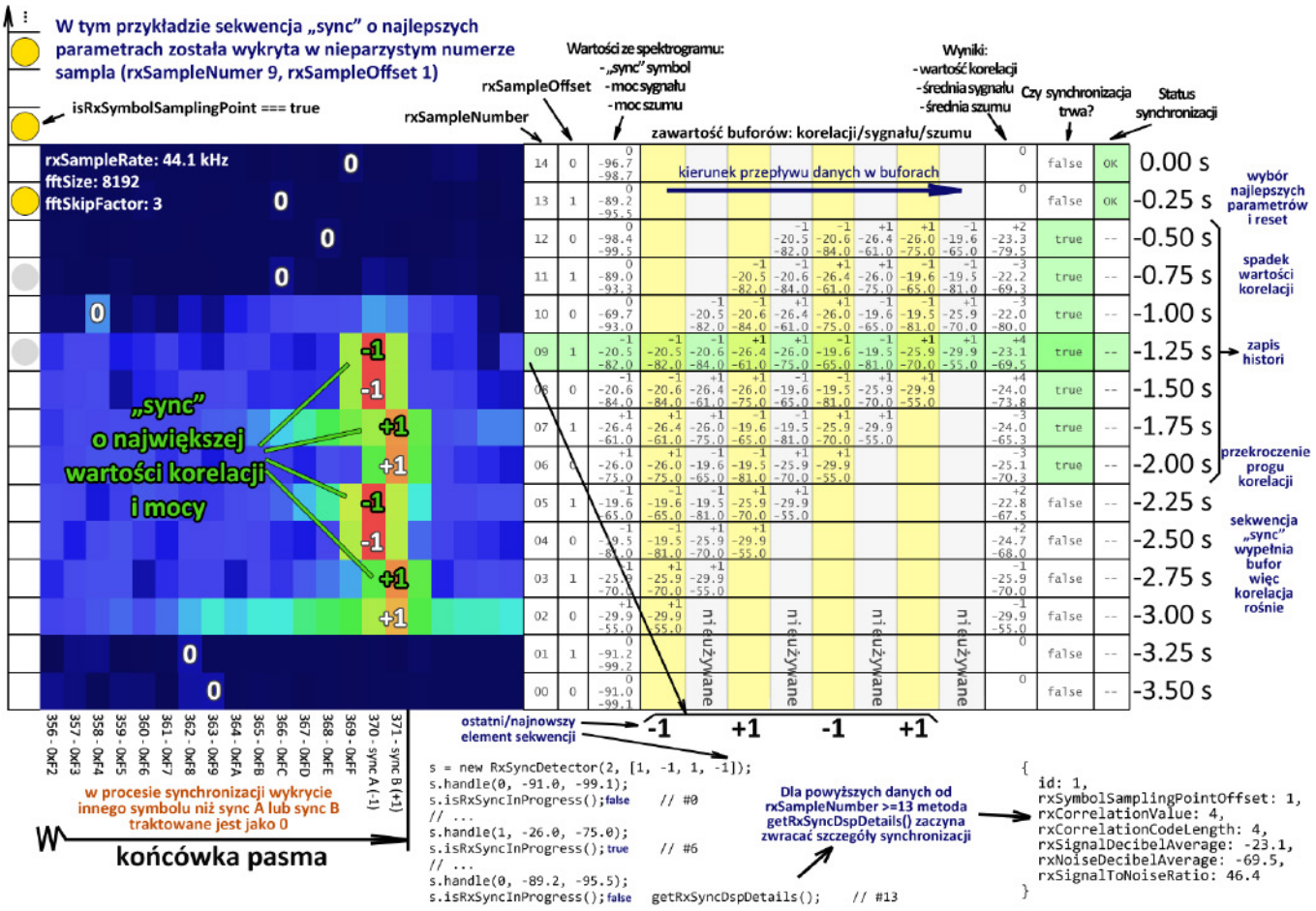
W naszej warstwie fizycznej także będziemy korzystać z wzorca naprzemiennych -1 oraz 1. Ten ciąg będziemy dalej nazywać sekwencją „sync". Do synchronizacji posłużymy się dwoma symbolami, dla których miejsce zarezerwowaliśmy na samym końcu naszego pasma. Odebranie przedostatniego symbolu (Sync A) zmapujemy na -1, ostatniego (Sync B) na 1, a dowolnie innego na 0.

Logikę wykrywającą sekwencję „sync" zawarto w klasie `RxSyncDetector`. Jej działanie przedstawiono na Rysunku 13.

» https://audio-network.rypula.pl/rx-sync-detector-class



Rysunek 12. Próbkowanie wyników FFT pozwalające otrzymać poprawne symbole



Rysunek 13. Szczegóły procesu synchronizacji

Jak widzimy, nasza klasa umożliwia obliczanie korelacji, patrząc na co drugi element. Umożliwia to naprzemienne przetwarzanie symboli o numerach parzystych, jak i nieparzystych w miarę ich napływu do bufora. Na początku, gdy sygnał „sync" nie jest transmitowany, korelacja jest zerowa. Gdy tylko nadajnik zacznie transmitować sekwencję „sync", współczynnik korelacji zacznie wzrastać. Od pewnego progu nasza klasa `RxSyncDetector` zaczyna zapisywać historię wyników – wartość współczynnika korelacji, średnią moc sygnału, średnią moc szumu oraz informacje o przesunięciu aktualnego symbolu. Metoda `isRxSyncInProgress` zaczyna zwracać wtedy `true`. Historia ta jest uzupełniana do momentu wykrycia pierwszego spadku współczynnika korelacji. Oznacza to, że nadajnik zwyczajnie skończył nadawać sekwencję „sync". Zapisana historia jest finalnie sortowana po wartości współczynnika korelacji oraz średniej mocy sygnału. Pierwszy element tak posortowanej tablicy zawiera informacje o sygnale „sync" o najlepszych parametrach.

Gdy proces dobiegnie końca, historia oraz bufory są czyszczone, a metoda `isRxSyncInProgress` zaczyna zwracać `false`. Szczegóły naszej synchronizacji możemy odczytać poprzez funkcję `getRxSyncDspDetails`. Oprócz wartości przesunięcia otrzymujemy także powiązaną z nim średnią wartość sygnału oraz średnią wartość szumu. Różnica tych wartości to SNR (ang. *signal to noise ratio*). Warstwa fizyczna mniej więcej w połowie wartości średnich (`rxSignalDecibelThresholdFactor`) ustala wartość progu mocy (`rxSignalDecibelThreshold`). Pozwala to odróżnić prawdziwy sygnał od szumu otoczenia. Wszystkie sygnały poniżej tego poziomu traktowane są jako brak aktywnej transmisji (ang. *idle line*).

Działanie synchronizacji możemy przetestować sami, korzystając z widocznego poniżej przykładu. Aby uwidocznić cały proces, pasmo celowo ograniczono do bardzo wąskiego zakresu znajdującego się tuż przy symbolach sync. „Surowe" sample tablic FFT pojawiają się od razu w odstępach co 250 ms. Nieco inaczej jest z zsynchronizowanymi symbolami, gdyż najpierw musimy odebrać przynajmniej jedną poprawną sekwencję „sync". Po jej wykryciu „synchronizowane" symbole zaczną pojawiać się w odstępach co 500 ms. Pamiętajmy jednak, że gdy moc sygnału takiego symbolu będzie poniżej progu, to zamiast wartości liczbowej otrzymamy wartość `null`.

» https://audio-network.rypula.pl/fsk-synchronization-demo
» https://audio-network.rypula.pl/fsk-synchronization-demo-src

WARSTWA FIZYCZNA (ANG. PHYSICAL LAYER)

W tej części opowiemy o naszej klasie `PhysicalLayer`. Spina ona zagadnienia DSP poruszone do tej pory. Z uwagi na dużą liczbę opcjonalnych parametrów, jakie musiałby przyjmować konstruktor, dla większej czytelności zdecydowano się zastosować wzorzec buildera. Prosty przykład przedstawiono w Listingu 5.

Listing 5. Przykład użycia klasy PhysicalLayer

```
// ...
function init() {
  physicalLayerBuilder = new PhysicalLayerBuilder();
```

```
  physicalLayer = physicalLayerBuilder
    .rxSymbolListener(rxSymbolListener)
    .rxSyncStatusListener(rxSyncStatusListener)
    // ... dostępne są też inne
    // listenery i opcje konfiguracji ...
    .build();

  // tą wartość musimy ustawić na drugim
  // urządzeniu metodą setTxSampleRate
  console.log(physicalLayer.getRxSampleRate());
}

function onSetTxSampleRateButtonClick() {
  var
    input = document.getElementById(
      'tx-sample-rate'
    ),
    txSampleRate = parseInt(input.value);

  // zmienna txSampleRate musi mieć wartość
  // odczytaną z drugiego urządzenia za pomocą:
  // physicalLayer.getRxSampleRate()
  physicalLayer.setTxSampleRate(txSampleRate);
}

function onTxSyncButtonClick() {
  physicalLayer.txSync();
}

function onTxByteButtonClick() {
  var
    input = document.getElementById('tx-byte'),
    byte = parseInt(input.value)
    txDspConfig = physicalLayer.getTxDspConfig(),
    txSymbol = txDspConfig.txSymbolMin + byte;

  physicalLayer.txSymbol(txSymbol);
}

function rxSymbolListener(o) {
  var rxDspConfig, byte, rxSymbolLog, rxByteLog;

  rxDspConfig = physicalLayer.getRxDspConfig();
  byte = o.rxSymbol
    ? o.rxSymbol - rxDspConfig.rxSymbolMin
    : null;
  rxSymbolLog = o.rxSymbol ? o.rxSymbol : 'idle';
  rxByteLog = byte !== null ? byte : '---';
  // ...
}

function rxSyncStatusListener(s) {
  var rxSyncOkLog, rxSyncInProgressLog;

  rxSyncOkLog = s.isRxSyncOk
    ? 'OK' : 'waiting for sync...';
  rxSyncInProgressLog = s.isRxSyncInProgress
    ? '[sync in progress]' : '';
  // ...
}
```

Pełny kod klasy `PhysicalLayer` możemy znaleźć poniżej:

» https://audio-network.rypula.pl/physical-layer-class

Dla poprawnej wymiany symboli kluczowa jest synchronizacja. Gdy na urządzeniu B wykonamy metodę `getRxSampleRate`, zwróci nam ona częstotliwość próbkowania, z jaką pracuje kontekst Web Audio API. Jest ona zależna od urządzenia i nie można jej zmienić. Wartość tą należy ustawić na urządzeniu A metodą `setTxSampleRate` oraz przesłać sekwencję „sync" metodą `txSync`. Gdy urządzenie B poprawnie ją zinterpretuje, konfiguracja transmisji w jedną stronę jest zakończona. Postępując analogicznie, skonfigurujemy transmisję w stronę przeciwną. Status synchronizacji możemy na bieżąco monitorować za pomocą listenera `rxSyncStatusListener`.

Poprawne odebranie sekwencji „sync" skutkuje rozpoczęciem cyklicznego wywoływania funkcji `rxSymbolListener`. Gdy drugie urządzenie nie transmituje żadnych danych, wtedy jako symbol będziemy otrzymywać wartość `null`. Do przesyłania danych należy posłużyć się metodą `txSymbol`. Po jej wywołaniu w listenerze `rxSymbolListener` po drugiej stronie powinniśmy otrzymać symbol, który nadaliśmy. Pojawi się on jednak tylko wtedy, gdy siła sygnału przekroczy próg mocy (`rxSignalDecibelThreshold`). Jest on ustawiany w momencie odebrania poprawnej sekwencji „sync" mniej więcej w połowie średniej wartości sygnału i szumu.

Pamiętajmy, że nasza klasa pracuje w trybie półdupleksu (ang. *half duplex*). Nie jest zatem możliwa transmisja w obie strony w tym samym czasie. Warte wspomnienia jest też to, że podczas aktywnej transmisji przetwarzanie próbek z mikrofonu jest blokowane tak, aby nadajnik nie odbierał symboli, które sam transmituje.

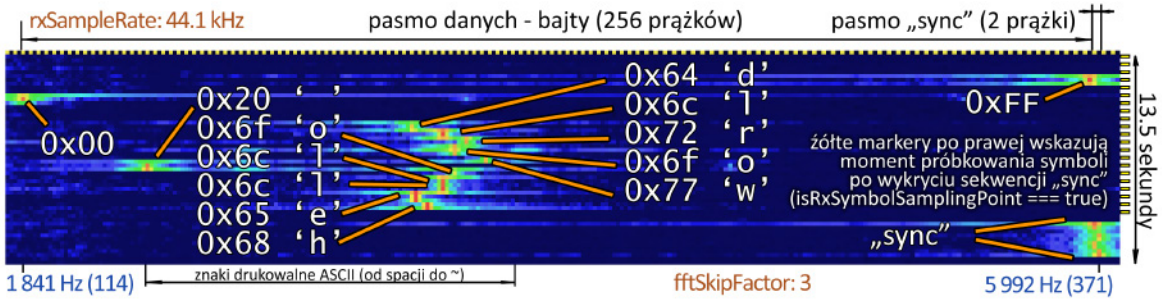
Musimy też pamiętać, że numery symboli wysyłanych i odbieranych to tak naprawdę numery prążków widmowych pasma, które wcześniej ustaliliśmy. W celu normalizacji na zakres rozpoczynający się od zera należy odjąć od numeru symbolu wartość `rxSymbolMin` lub dodać `txSymbolMin`. Z kolei te wartości możemy odczytać z obiektu zwracanego przez metody `getRxDspConfig` bądź `getTxDspConfig`. Wybór właściwej metody zależy od tego, czy konwertujemy symbol odbierany, czy nadawany (Listing 5). W naszym paśmie przewidziano 256 symboli na dane, więc po normalizacji otrzymamy zakres jednego bajta (0x00 - 0xFF).

Testy we własnym zakresie możemy wykonać sami, korzystając z poniższych przykładów:

» https://audio-network.rypula.pl/physical-layer-simplest
» https://audio-network.rypula.pl/physical-layer-simplest-src
» https://audio-network.rypula.pl/physical-layer-simple
» https://audio-network.rypula.pl/physical-layer-simple-src

Listener `rxSymbolListener` jest jednym z wielu, jakie możemy zarejestrować. Na przykład szczegóły związane m.in. z mocą odbieranego sygnału możemy odczytać za pomocą listenera `rxSampleDspDetailsListener`, który wywoływany jest dwukrotnie częściej niż `rxSymbolListener`. Z jego pomocą możemy np. wygenerować jeden wiersz spektrogramu.

Na Rysunku 14 pokazano, jak zmienia się widmo amplitudowe w miarę upływu czasu podczas przesyłania bajtów wiadomości „hello world".



Rysunek 14. Spektrogram przedstawiający transmisję wiadomości „hello world" w warstwie fizycznej

Przykład demonstrujący wszystkie możliwe listenery i zwracane przez nie dane znajdziemy poniżej:

» https://audio-network.rypula.pl/physical-layer-listeners-demo
» https://audio-network.rypula.pl/physical-layer-listeners-demo-src

Do kompletu dostępny jest także przykład umożliwiający przesyłanie znaków ze zbioru ASCII:

» https://audio-network.rypula.pl/physical-layer
» https://audio-network.rypula.pl/physical-layer-src

WARSTWA ŁĄCZA DANYCH (ANG. DATA LINK LAYER)

Gdy odbierany sygnał jest dostatecznie silny, a zakłócenia znikome, nasza warstwa fizyczna działa w zasadzie bezbłędnie. Niestety problem pojawia się od razu wtedy, gdy moc innego dźwięku przekroczy moc sygnału z nadajnika. Musimy zatem stworzyć pewien mechanizm kontroli błędów.

Rozwiązaniem jest tutaj „paczkowanie" strumienia danych w ramki z nagłówkiem oraz sumą kontrolną. Zadaniem odbiornika jest zatem ciągłe buforowanie napływających bajtów i przeliczanie sumy kontrolnej z bajtów mogących potencjalnie zawierać ramkę. Rozwinięciem tego rozwiązania byłoby użycie zamiast sumy kontrolnej systemu korekcji błędów. Przykładem może być np. kodowanie Reeda-Solomona. Zredukowałoby to liczbę odrzuconych ramek. My jednak dla uproszczenia poprzestaniemy na rozwiązaniu z bardzo prostą w implementacji sumą kontrolną. Propozycję wyglądu kompletnej ramki danych przedstawiono na Rysunku 15.

Jak widzimy, na początku pierwszego bajta będącego nagłówkiem ramki znajdują się 3 jedynki pod rząd. Jest to nasz marker początku ramki. Następny bit to flaga mówiąca o tym, czy dane zawarte w ramce należy traktować jako komendę warstwy łącza danych, czy zwykłe dane. Do komend wrócimy nieco później. Kolejny bit (`isOfdm`) jest nieużywany. Zostawiono go na poczet przyszłościowej obsługi ramek zawierających symbole OFDM. Bit ten jest naszą furtką do dalszej rozbudowy, o której mówiliśmy przy okazji metod modulacji. Ostatnie 3 bity pierwszego oktetu reprezentują liczbę bajtów ładunku (ang. *payload*). Ładunkiem mogą być bajty danych bądź komendy. Liczba bajtów ładunku waha się od 0 do 7 (2^3-1). Ostatni oktet ramki to suma kontrolna obliczona ze wszystkich poprzednich bajtów, włączając nagłówek. Do jej realizacji wybrano bardzo prosty algorytm Fletchera w wersji 8 bitowej (Listing 6).

Listing 6. Ośmiobitowa suma kontrolna Fletchera

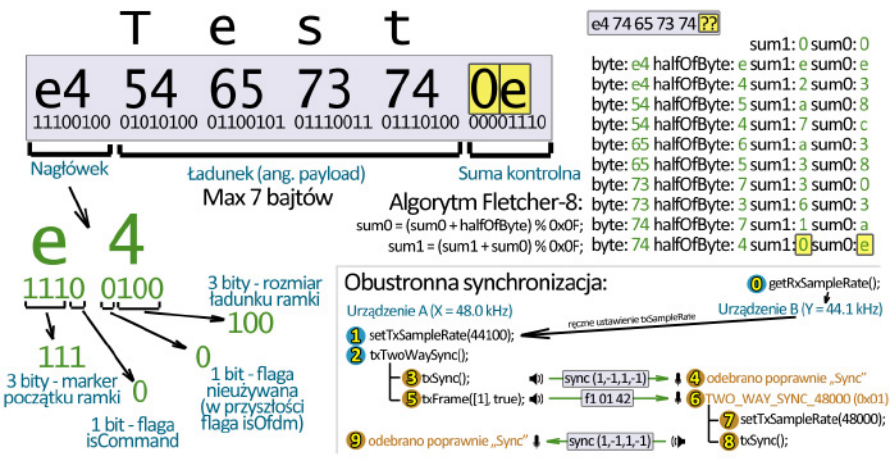
```
function computeFletcher8Checksum(data) {
  var
    sum0, sum1, i, isLeftHalfOfByte,
    byteNumber, byte, halfOfByte;

  sum0 = 0;
  sum1 = 0;
  for (i = 0; i < 2 * data.length; i++) {
    isLeftHalfOfByte = i % 2 === 0;
    byteNumber = i >>> 1;
    byte = data[byteNumber];
    halfOfByte = isLeftHalfOfByte
      ? (byte & 0xF0) >>> 4
      : byte & 0x0F;
    sum0 = (sum0 + halfOfByte) % 0x0F;
    sum1 = (sum1 + sum0) % 0x0F;
  }

  return (sum1 << 4) | sum0;
}
```

Nasza warstwa fizyczna w listenerze ustawionym metodą `rxSymbolListener` dostarcza symbol FSK przepuszczony wcześniej przez test progu mocy (`rxSignalDecibelThreshold`). Jest on ustawiany na podstawie poprawnie odebranej sekwencji „sync". Na poziomie warstwy fizycznej jest to jedyne kryterium rozróżniające sygnał od szumu. Warstwa łącza danych sprawdza jednak integralność ramki, korzystając z sumy kontrolnej. Możemy jej zatem z powodzeniem szukać w „surowych" symbolach, które mogą być także szumem. Pozwoli to na odbieranie danych nawet o bardzo małej mocy sygnału.

Algorytm „wypatrywania" ramek jest bardzo prosty. Wystarczy każdy „surowy" bajt porównać ze wzorcem nagłówka ramki (jedynki w 3 bitach od lewej oraz zero w bicie nieużywanym). Gdy znajdziemy taki bajt, odczytujemy z niego długość ładunku. To z kolei pozwala określić, ile bajtów należy zbuforować, by finalnie odebrać bajt sumy kontrolnej. Weryfikacja poprawności odebranych bajtów sprowadza się do samodzielnego obliczenia z nich sumy kontrolnej i porównanie jej z tą odebraną. Jeśli bajty są sobie równe, jest to znak, że odebraliśmy poprawną ramkę.

W danej chwili kandydatów na ramkę może być wiele. Zwyczajnie nowy marker możemy znaleźć w środku innej potencjalnej ramki. Do momentu odebrania sumy kontrolnej nie jesteśmy jednak w stanie stwierdzić, który kandydat jest poprawny. Listę aktualnych kandydatów możemy odczytać za pomocą listenera `rxFrameCandidateListener`. Gdy któraś z potencjalnych ramek okaże się poprawną, zostaje przekazana do drugiego listenera o nazwie `rxFrameListener`. Gdy w tym samym czasie więcej niż jeden kandydat okaże się poprawną ramką, wybieramy tą najdłuższą.

Do poprawnej wymiany ramek urządzenia muszą być ze sobą synchronizowane w warstwie fizycznej (obustronne ustawienie `sampleRate` oraz przesłanie sekwencji „sync"). Warstwa łącza danych upraszcza nieco ten proces, gdyż połowa konfiguracji wykonywana jest za nas automatycznie. W dalszym ciągu musimy na początku ustawić wartość `rxSampleRate`, lecz zamiast wywołać metodę `txSync`, użyjemy `txTwoWaySync`. Wykorzystuje ona możliwość przesyłania komend. Jej wywołanie przez urządzenie A spowoduje przesłanie



Rysunek 15. Ramka warstwy łącza danych oraz szczegóły procesu obustronnej synchronizacji

sekwencji „sync" oraz ramki z komendą `TWO_WAY_SYNC_44100` (payload 0x00) lub `TWO_WAY_SYNC_48000` (payload 0x01). Rodzaj komendy zależy od `sampleRate` urządzenia A. Odebranie takiej ramki na urządzeniu B skutkuje zdalnym wykonaniem metod `setTxSampleRate(x)` oraz `txSync`. Parametr x wnioskowany jest w zależności od odebranej komendy – może to być 44100 bądź 48000.

Innymi słowy urządzenie A mówi: „Hej, wiem, że używasz `sampleRate` Y Hz, ponieważ użytkownik ręcznie ustawił mi tą wartość. Wysyłam Ci moją sekwencję sync, byś mógł odbierać ode mnie ramki danych. Chwilę potem wysyłam Ci ramkę z komendą, z której wywnioskujesz wartość mojego `rxSampleRate` równą X Hz. Ustaw sobie taką wartość i też wyślij mi sekwencję sync, bym także mógł odbierać dane od Ciebie" (Rysunek 15). Trochę naciąganą analogią z życia codziennego może być łączenie się do sieci Wi-Fi i negocjacja parametrów łącza radiowego.

W bardzo prosty sposób można by w ogóle pominąć początkowy krok ustawiania wartości `txSampleRate`. Wystarczyłoby zapoczątkować synchronizację od 44100 Hz i dopiero w przypadku braku odpowiedzi wykonać próbę dla 48000 Hz. Gdy jednak nie trafimy za pierwszym razem, cały proces wydłuży się o kilka sekund. Z uwagi na tę wadę porzucono ten pomysł.

Zainteresowanych szczegółami implementacji powyższych rozwiązań odsyłam do analizy klasy `DataLinkLayer`, której kod znajdziemy poniżej:

» https://audio-network.rypula.pl/data-link-layer-class

W Listingu 7 przedstawiono jeden z prostszych sposobów użycia naszej klasy.

Listing 7. Przykład użycia klasy DataLinkLayer

```
// ...
function init() {
  dataLinkLayerBuilder = new DataLinkLayerBuilder();
  dataLinkLayer = dataLinkLayerBuilder
    .rxFrameListener(rxFrameListener)
    // ...
    .build();

  // do ustawienia na drugim urządzeniu
  console.log(dataLinkLayer.getRxSampleRate());
}

function rxFrameListener(frame) {
  console.log(frame.isCommand); // np: false
  console.log(frame.payload);  // np: [0x61, 0x62, 0x63]
}

function onSetTxSampleRateButtonClick() {
  // wartość odczytana z drugiego urządzenia
  // ...
  dataLinkLayer.setTxSampleRate(txSampleRate);
}

function onTxTwoWaySyncButtonClick() {
  dataLinkLayer.txTwoWaySync();
}

function onTxAsciiTestButtonClick() {
  var
    payload = [0x61, 0x62, 0x63], // 'abc'
    isCommand = false,

  dataLinkLayer.txFrame(payload, isCommand);
}
```

Linki do przykładów znajdziemy poniżej:

» https://audio-network.rypula.pl/data-link-layer
» https://audio-network.rypula.pl/data-link-layer-src

Wnikliwi czytelnicy zapewne zauważą, że poprawne ramki można by odbierać także bez wcześniejszej synchronizacji. W warstwie łącza danych zamiast opierać się na „zsynchronizowanych" symbolach dostarczanych co 500 ms wystarczyłoby przetwarzać dwa strumienie surowych symboli FSK przesunięte o 250 ms. Jeden pochodziłby z nieparzystych, a drugi z parzystych numerów tablic wyników FFT. Takie rozwiązanie mogłoby powodować powstawanie duplikatów ramek, które jednak dość łatwo można by usuwać. Jak widzimy, pole rozwoju naszego systemu jest duże. Z uwagi jednak na przyjęcie zasady KISS, przynajmniej na początku, nasz system opiera się na synchronizacji zaimplementowanej w warstwie fizycznej.

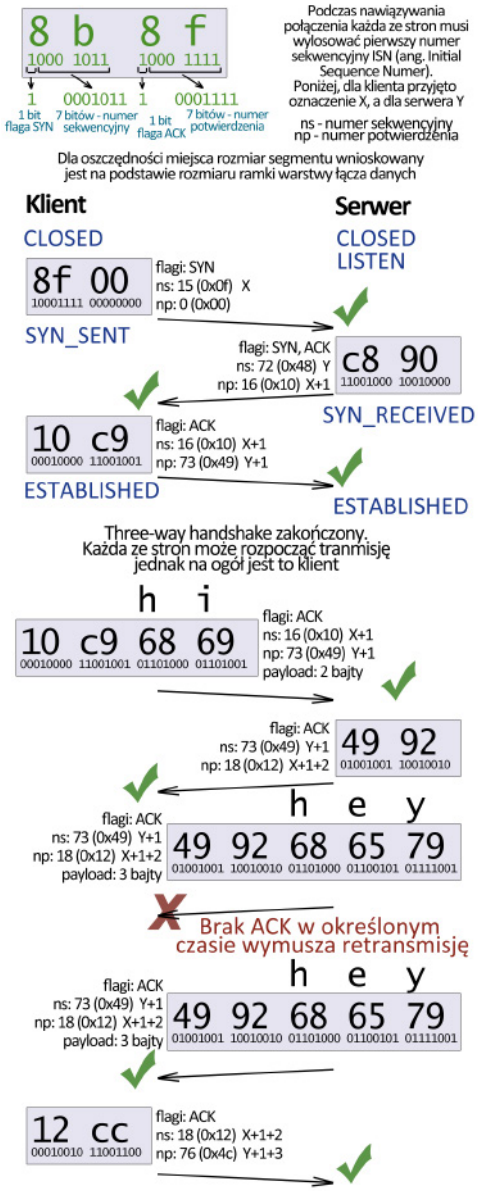
WARSTWA TRANSPORTOWA (ANG. TRANSPORT LAYER)

Użycie warstwy łącza danych pozwala wykryć błędy transmisji. Niestety w przypadku wystąpienia błędu zwyczajnie zostajemy z „dziurą" w naszym strumieniu bajtów. By rozwiązać ten problem, nasza warstwa transportowa opiera się na uproszczonej formie protokołu TCP (ang. *Transmission Control Protocol*). Obsługiwane jest tylko jedno połączenie bez portów. Zestawianie połączenia opiera się na metodzie *three-way handshake*. Każde urządzenie musi zatem przechowywać swój stan połączenia. W przypadku błędu dopiero kilkukrotna próba retransmisji skutkuje zerwaniem połączenia. Dla uproszenia w naszej implementacji pominięto wiele elementów obecnych w TCP, m.in. proces zamykania połączenia czy zmiana rozmiaru okna.

Do poprawnej pracy naszego uproszczonego protokołu TCP konieczna jest wcześniejsza obustronna synchronizacja urządzeń, tak by mogły między sobą przesyłać ramki danych. O tym, jak to zrobić, pisaliśmy przy okazji omawiania wcześniejszych warstw. Obustronna komunikacja jest kluczowa z uwagi na konieczność potwierdzania dostarczenia segmentów. Kolejnym krokiem jest ustalenie, które urządzenie jest serwerem, a które klientem. Na samym początku oba urządzenia są w stanie `CLOSED`. Aby serwer mógł przyjąć połączenie, musi przejść w stan `LISTEN`. Wtedy rolą klienta jest zainicjalizowanie procesu zestawiania połączenia poprzez rozpoczęcie procedury *three-way handshake*.

Kontrola spójności strumienia bajtów opiera się na numerach sekwencyjnych oraz numerach potwierdzenia. Na początku każda ze stron musi wylosować początkowy numer sekwencyjny (ang. *ISN, Initial Sequence Number*), który musi przesłać stronie przeciwnej. Sygnalizuje to aktywna flaga `SYN` (ang. *synchronize*). Gdy flaga `SYN` jest nieustawiona, wtedy numer sekwencyjny mówi o pozycji danych zawartych w segmencie w całym strumieniu bajtów jednej ze stron danego połączenia. Ładunek (ang. *payload*) segmentu zwiększa zatem ten numer o liczbę bajtów w nim zawartych. Przyjęto, że segment SYN, choć nie zawiera bajtów danych, również zwiększa numer sekwencyjny, lecz jedynie o jeden. Numer potwierdzenia pozwala natomiast określić, ile bajtów danych zostało poprawnie odebranych. Ważność tego pola sygnalizuje ustawiona flaga ACK (ang. *acknowledgment*).

Dla uproszczenia naszej implementacji każdy wysłany segment z danymi wymaga zwrotnego segmentu z potwierdzeniem. W dość łatwy sposób można by usprawnić transmisję, umożliwiając wysyłanie ustalonej wcześniej liczby segmentów w ciągu, jeden po drugim. Segment zwrotny informowałby nas, do jakiej pozycji otrzymano dane bez wystąpienia luki. Takie rozwiązanie zmniejszyłoby narzut związany z ciągłym przesyłaniem potwierdzeń. Podobny mecha-



Rysunek 16. Szczegóły budowy segmentu oraz przykład transmisji uproszczonego protokołu TCP

nizm zaimplementowano w prawdziwym protokole TCP. Po więcej szczegółów odsyłam do źródeł zewnętrznych.

Nagłówek naszego uproszczonego protokołu TCP składa się z dwóch bajtów. Pierwszy zawiera flagę SYN oraz numer sekwencyjny. Drugi flagę ACK oraz numer potwierdzenia. Każdy z numerów dysponuje zatem 7 bitami do zapisania wartości będącej liczbą całkowitą bez znaku. Mamy zatem do dyspozycji wartości od 0 do 127. Przekręcenie licznika powoduje powrót na początek zakresu.

Z uwagi na oszczędność miejsca w nagłówku nie przewidziano pola na rozmiar ładunku. Ta wartość jest ustalana na podstawie rozmiaru ramki warstwy łącza danych, w której segment podróżuje. W niej z kolei można zmieścić 7 bajtów ładunku. Oznacza to, że maksymalna liczba bajtów danych użytkowych segmentu wynosi 5. Dwa pozostałe bajty tracimy na nagłówek.

Strukturę segmentu oraz przykład komunikacji rozjaśniający opisywane zagadnienia przedstawiono na Rysunku 16.

Gdy obie strony przejdą w stan `ESTABLISHED`, wtedy każda z nich może rozpocząć transmisję. Musimy jednak pamiętać, że warstwa fizyczna działa w trybie półdupleksu, więc nie jest możliwa transmisja w obie strony naraz. Gdy o tym zapomnimy, nastąpi kolizja, gdyż oba urządzenia pracują w pokrywającym się paśmie.

Całość warstwy transportowej zamknięto w klasie `TransportLayer`, której kod źródłowy znajdziemy poniżej:

» https://audio-network.rypula.pl/transport-layer-class

Przykład prezentujący działanie naszej klasy dostępny jest poniżej:

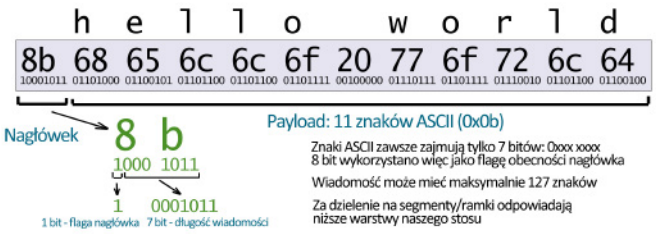
» https://audio-network.rypula.pl/transport-layer
» https://audio-network.rypula.pl/transport-layer-src

WARSTWA APLIKACJI (ANG. APPLICATION LAYER) – PROTOKÓŁ WYMIANY TEKSTU

Warstwa aplikacji jest ostatnią, jakiej potrzebujemy do realizacji prostego komunikatora. Instancję klasy `TransportLayer` możemy traktować jako gniazdko sieciowe. Pisząc do niego strumień bajtów, możemy mieć pewność, że nasze bajty będą sukcesywnie pojawiać się po stronie odbiornika w niezmienionej kolejności. W przypadku błędu nastąpi kilkukrotna retransmisja. W najgorszym przypadku nasze połączenie zostanie zerwane, o czym zostaniemy poinformowani. To, czego ciągle nam brakuje, to sposób oddzielania wiadomości od siebie. Do zrealizowania prostego chata musimy więc wybrać odpowiedni protokół wymiany danych.

Użytkownik korzystając z naszej aplikacji, przesyłać będzie krótkie wiadomości tekstowe. Dla uproszczenia korzystać będziemy jedynie ze zbioru ASCII. Wybierając odpowiedni dla nas protokół, musimy pamiętać o ograniczeniach przepustowości. Użycie HTTP byłoby ekstremalnie złym pomysłem, gdyż większość bajtów zmarnowalibyśmy na nagłówkach.

Tak naprawdę jedyne, czego potrzebujemy, to podanie długości wiadomości z jednoczesnym zaznaczeniem granicy pomiędzy nimi. Rozwiązaniem jest umieszczenie tej informacji w jednobajtowym nagłówku umieszczonym przed każdym blokiem tekstu. Użycie kodu ASCII gwarantuje, że bajty wiadomości są mniejsze od 128. By odróżnić bajt nagłówka od bajtów tekstu, należy zatem do 7 bitowej wartości długości wiadomości dodać 128 (0x80). Ustawi to najbardziej znaczący bit w bajcie. Pojedyncza wiadomość może mieć zatem długość od 0 do 127 znaków. Szczegóły naszego protokołu przedstawiono na Rysunku 17.



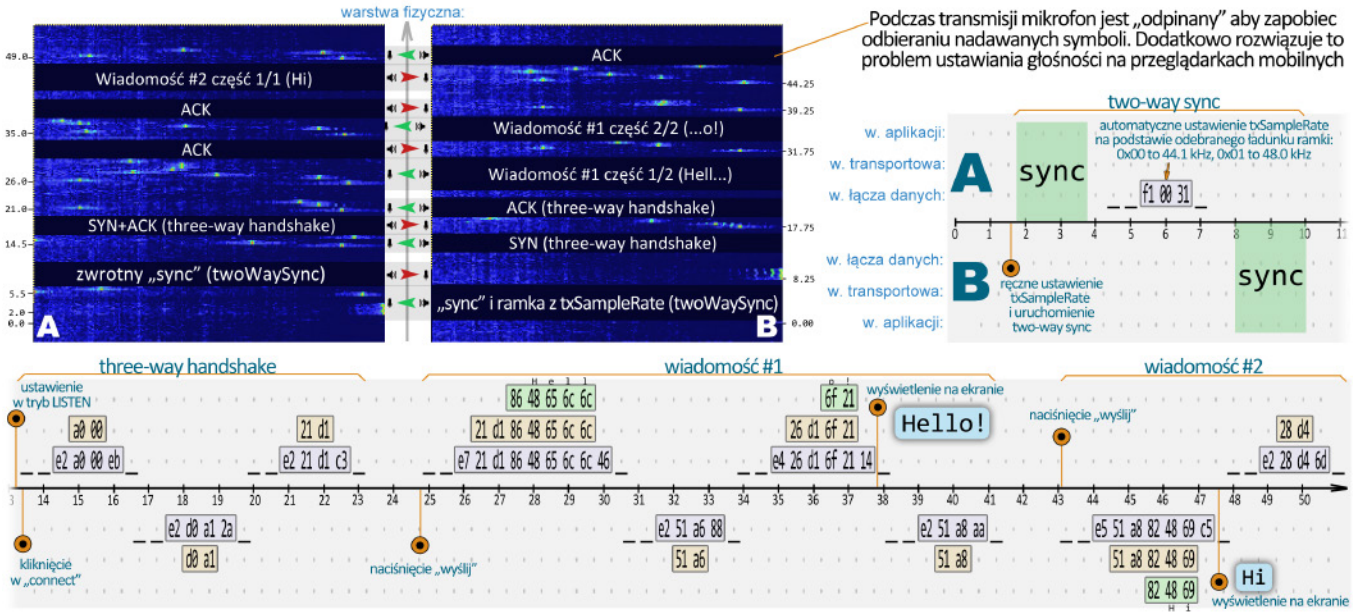
Rysunek 17. Prosty protokół wymiany tekstu

APLIKACJA AUDIO CHAT

Zwieńczeniem naszej przeprawy przez wszystkie warstwy sieciowe jest aplikacja pozwalająca przesyłać wiadomości tekstowe między dwoma urządzeniami za pomocą dźwięku:

» https://audio-network.rypula.pl/audio-chat
» https://audio-network.rypula.pl/audio-chat-src

Sprawdźmy zatem, co dzieje się w naszym powietrzu podczas przesyłania dwóch przykładowych wiadomości (Rysunek 18).



Rysunek 18. Aplikacja Audio Chat, czyli nasz stos sieciowy w akcji

Analiza czasów przesyłania wiadomości 1 i 2 pozwala stwierdzić, że z początkowych 2 bajtów na sekundę (warstwa fizyczna) realnie jesteśmy w stanie uzyskać prędkości rzędu połowy bajta na sekundę (warstwa aplikacji).

Na zmniejszenie efektywnej prędkości transmisji wpływa w pewnej mierze problem ustawiania głośności na niektórych przeglądarkach mobilnych. Często zamiast paska głośności mediów pojawia się pasek głośności rozmowy, o czym pisaliśmy przy okazji omawiania spektrogramu. Testy pokazały, że w takich przypadkach zmiana głośności nie zawsze będzie działać prawidłowo, co może powodować np. generowanie dźwięku o dużej mocy, nawet gdy pasek ustawiony jest na minimum. Zainteresowanych szczegółami tego problemu odsyłam do poniższej strony:

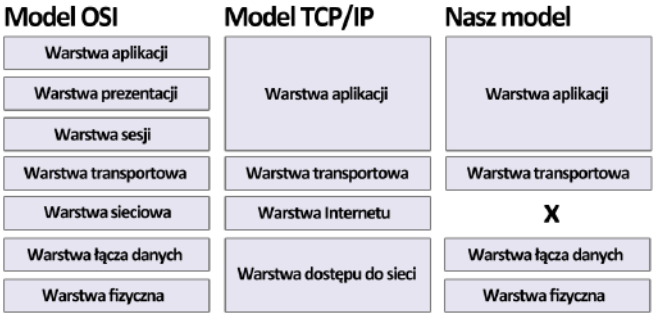
» https://bugs.chromium.org/p/chromium/issues/detail?id=326920

Rozwiązaniem jest programowe „odpięcie" się od mikrofonu na czas transmisji i ponowne „przypięcie" po jej zakończeniu. Implementacje odpowiednich metod znajdziemy w naszej klasie `AudioMonoIO`. Po takim zabiegu Web Audio API przepina się na inny strumień audio i na czas transmisji mamy dostęp do normalnego paska głośności mediów. To jednak nie wszystko. Zmiany konfiguracji nie powinno się dokonywać tuż przed lub tuż po zakończeniu generowania dźwięku, gdyż może to powodować powstawanie trzasków. Dodatkowo proces przypinania się do mikrofonu nie jest natychmiastowy. Dla bezpieczeństwa konieczne jest więc dodanie pustych symboli FSK generujących ciszę.

Opisany wyżej problem występuje jednak tylko na niektórych przeglądarkach mobilnych. Transmitując dane pomiędzy dwoma komputerami lub laptopami, moglibyśmy zredukować odstępy między ramkami do minimum, zwiększając przy tym efektywną prędkość transmisji.

PORÓWNANIE Z MODELEM OSI ORAZ TCP/IP

To, co zrealizowaliśmy w tej części artykułu, jest minimalistyczną implementacją pewnego miksu modelu OSI oraz TCP/IP (Rysunek 19). Z założenia system miał działać jedynie w obrębie dwóch urządzeń, z których tylko jedno może nadawać w danej chwili. Nie było więc potrzebne żadne adresowanie w warstwie łącza danych (adresy MAC). Z tego samego powodu zrezygnowaliśmy w całości z warstwy internetowej (ang. *Internet Layer*) oraz adresów IP. Warstwa transportowa została pozbawiona obsługi portów oraz całej masy flag i parametrów. Jedyne, co pozostawiono z TCP, to nawiązywanie połączenia oraz potwierdzanie odbioru segmentów.



Rysunek 19. Porównanie naszego stosu sieciowego z modelem OSI oraz TCP/IP

Eliminacja wszystkich zbędnych nam rzeczy pozwoliła znacznie uprościć cały system oraz oszczędzić całą masę bajtów narzutu. Jak widzimy, nawet tak odchudzony system ciągle umożliwia stworzenie praktycznej aplikacji.

JAK ROZBUDOWAĆ NASZ SYSTEM I ZWIĘKSZYĆ PRĘDKOŚĆ TRANSMISJI?

Trzeba przyznać szczerze, że prędkość 2 bajtów na sekundę, z jaką pracuje nasza warstwa fizyczna, nie należy do zawrotnych. Rzutuje to na wszystkie wyższe warstwy naszego stosu sieciowego. Co można zatem zrobić, by zwiększyć prędkość?

Rozwiązaniem jest skorzystanie z techniki OFDM oraz modulacji PSK lub nawet QAM. Specjalnie w tym celu w nagłówku ramki warstwy łącza danych zostawiliśmy jeden bit flagi `isOfdm`. Jego ustawienie może zwyczaje włączyć inną logikę przetwarzania symboli ładunku ramki (ang. *payload*). Na przykład zamiast 7 bajtów ukry-

tych w symbolach FSK możemy przetwarzać 7 różnych zakresów o długości 8192 próbek, w których znajdować się będzie wiele takich samych symboli OFDM o długości np. 128 próbek (Rysunek 6). Uśredniając fazy i amplitudy ze wszystkich symboli OFDM z danego zakresu, powinniśmy otrzymać bardzo przyzwoite punkty na diagramie konstelacji. Ostatni bajt ramki, czyli sumę kontrolną, możemy już odebrać tak jak nagłówek, czyli jako zwykły symbol FSK.

Używając w symbolu OFDM jedynie 4 podnośnych (dane) oraz modulując je metodą QPSK (inaczej 4-PSK, 2 bity na nośną), uzyskalibyśmy w łatwy sposób 1 bajt. Jest to dokładnie tyle, ile uzyskaliśmy w rozwiązaniu wykorzystującym metodę FSK opisaną w tym artykule. Zwiększenie tej liczby sprowadza się do zmiany typu modulacji lub/i liczby nośnych. Na przykład modulacja metodą 16-PSK (4 bity na nośną) oraz użycie 8 nośnych dałoby w rezultacie 4x większą prędkość transmisji. Pamiętajmy jednak, że jednym ze sposobów praktycznej realizacji OFDM jest dodanie kilku dodatkowych podnośnych pilotażowych o znanej fazie.

Tutaj pozwolimy sobie na mała dygresję. W wielu technologiach radiowych liczba podnośnych jest stała. Wtedy na prędkość transmisji w dużej mierze wpływa typ modulacji. Im sygnał jest silniejszy, tym wyższy schemat modulacji możemy użyć w podnośnych. W świecie radiowym niczym niezwykłym jest stosowanie np. modulacji 256-QAM (8 bitów na nośną). Gdy siła sygnału spada, wybierane są coraz wolniejsze metody modulacji, np. 64-QAM (6 bitów/nośną), 16-QAM (4 bity/nośną), QPSK (2 bity/nośną) lub BPSK (1 bit/nośną). Przekładając to na życie codziennie, między innymi dlatego, gdy oddalamy się z laptopem od routera Wi-Fi, nasza prędkość łącza stopniowo spada.

Ok, wróćmy do audio. W naszym przypadku, aby nieco ograniczyć liczbę próbek, jakie musimy przetwarzać, możemy pokusić się o samodzielną redukcję częstotliwości próbkowania naszego sygnału. Pasmo, jakie używa nasz system, jest zwyczajnie małym wycinkiem pełnego zakresu częstotliwości, jakie dostarcza Web Audio API. Aby zrealizować nasz pomysł, musielibyśmy wcześniej użyć miksera, który przesunie nasze pasmo do niższych częstotliwości, oraz filtra dolnoprzepustowego, by uniknąć zjawiska aliasingu. Wtedy moglibyśmy przetwarzać np. co czwartą próbkę pierwotnego sygnału bez utraty informacji (ang. *decimation*). Jeszcze lepszym rozwiązaniem byłby resampling na częstotliwość próbkowania wspólną dla wszystkich urządzeń, np. 16384 Hz. Wtedy nie byłoby konieczne ustawianie wartości `sampleRate` za pomocą pary metod `getRxSampleRate`, `setTxSampleRate`.

Idąc dalej, możemy pokusić się o własną implementację algorytmu FFT. Umożliwiłoby to szybkie przetwarzanie dużej liczby podnośnych wraz z informacją o przesunięciu fazowym. Co ciekawe, najprostsza implementacja FFT w formie rekurencji mieści się w kilkudziesięciu linijkach kodu.

W tą właśnie stronę przyspieszenia transmisji zmierza mój hobbystyczny projekt Audio Network rozwijany równolegle z tą serią artykułów. Jak dotąd testy praktyczne części z powyższych technik DSP dały obiecujące rezultaty. Przysłowiową „wędką" dla innych niech będą poniższe przykłady:

- https://audio-network.rypula.pl/fir-filter
- https://audio-network.rypula.pl/fir-filter-src
- https://audio-network.rypula.pl/fft
- https://audio-network.rypula.pl/fft-src
- https://audio-network.rypula.pl/iq-mixer
- https://audio-network.rypula.pl/iq-mixer-src

Wszystkich zainteresowanych szczegółami techniki OFDM, filtrami FIR (ang. *Finite Impulse Response*), miksowaniem IQ czy też FFT odsyłam do źródeł zewnętrznych. Jest to szeroki temat wykraczający poza zakres tego artykułu. Dodatkowo polecam zapoznać się z poniższą stroną. Jest to zbiór dokumentów, postów oraz filmów o tematyce DSP, jakie okazały się pomocne podczas tworzenia tej serii.

- https://audio-network.rypula.pl/internet-sources

PODSUMOWANIE

Ta część kończy serię artykułów o transmisji danych dźwiękiem. Rozpoczęliśmy ją od podstaw Cyfrowego Przetwarzania Sygnałów, poznając intuicyjny i prosty algorytm Dyskretnej Transformaty Fouriera. Potem wzięliśmy pod lupę Web Audio API, co pozwoliło nam przetestować nasze pomysły w praktyce. Finalnie zrealizowaliśmy prosty stos sieciowy bazujący na mieszance modelu OSI oraz TCP/IP. Pozwoliło to stworzyć aplikację „Audio Chat", w której wiadomości otrzymują potwierdzenie dostarczenia, a warstwa transportowa naszego stosu sieciowego dokonuje retransmisji w przypadku błędu.

Trudno wyobrazić sobie dzisiejszy świat bez elektronicznych metod wymiany informacji. Prawdziwy rozkwit przeżywają teraz radiowe technologie mobilne, oferując coraz to większe prędkości transmisji. Weszły już one tak mocno w nasze życie, że większość z nas nawet nie zastanawia się, jak to się dzieje, że nasze telefony są w stanie przesyłać dane bez użycia kabla. Oczywiście prędkość działania naszego prostego systemu audio trudno porównywać z technologiami radiowymi. W obydwu jednak przypadkach mamy do czynienia z falami, a podstawy DSP są podobne. Oznacza to, że nawet nasza klasa `WaveAnalyser` z powodzeniem byłaby w stanie przetwarzać próbki sygnału radiowego.

Radio definiowane programowo nie jest niczym nowym i znane jest pod nazwą SDR (ang. *Software Defined Radio*). Za około 20 dolarów możemy kupić urządzenie przypominające wielkością pendrive'a. Umożliwia ono podglądanie spektrum radiowego w zakresie od kilkudziesięciu MHz do prawie 2 GHz. Darmowe oprogramowanie umożliwia nagranie wycinka spektrum do postaci tablicy próbek. Nic zatem nie stoi na przeszkodzie, by w JavaScripcie przetwarzać sygnał naszego pilota do bramy garażowej lub dekodować naszą ulubioną stację radiową FM wraz z nazwą aktualnej piosenki ukrytą w sygnale RDS.

Jak widzimy, możliwości DSP są ogromne. Seria tych artykułów, jak i projekt Audio Network powstały z czystej ciekawości i chęci znalezienia odpowiedzi na pytanie, jak to możliwe, że ze strumienia próbek można wyciągnąć informacje o zawartych w nim częstotliwościach i dodatkowo odczytać z nich użyteczne dane. Oczywiście to, o czym mówiliśmy, jest tylko wycinkiem tego, co można zrealizować za pomocą technik DSP. Myślę jednak, że mając opanowane podstawy, dużo łatwiej jest szukać nowych informacji.

ROBERT RYPUŁA
robert.rypula@gmail.com

Pasjonat komputerów i programowania. Od wielu lat związany z aplikacjami webowymi. Obecnie zatrudniony w PGS Software na stanowisku Frontend Developer. Wcześniej doświadczenie zdobywał w firmie Okinet. Poza technologiami webowymi twórca m.in. aplikacji HgtReader umożliwiającej rendering topografii całej Ziemi (OpenGL/Qt). Fan pisania własnych rozwiązań od zera wszędzie tam, gdzie jest to możliwe.