AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA TELEKOMUNIKACJI

PRACA DYPLOMOWA INŻYNIERSKA / MAGISTERSKA

Implementacja techniki SIMO w programowym dekoderze cyfrowego radia DAB+

Implementation SIMO technique in digital software radio decoder DAB+

Autor: *Michał Rzepka*
Kierunek studiów: *Elektronika i Telekomunikacja*
Opiekun pracy: *dr inż. Jarosław Bułat*

Kraków, 2018

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „ Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej „sądem koleżeńskim”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

……

Spis Treści

Wstęp

Postęp technologiczny ma znaczenie w życiu każdego człowieka. Nowe rozwiązania ułatwiają nam wykonywanie codziennych czynności oraz zapewniają rozrywkę. Telewizja i radio to technologie, z którymi człowiek ma do czynienia na co dzień. Obie gałęzie wciąż ulegają ewolucji. W Polsce w ostatnich latach doświadczyliśmy kolejnego etapu rozwoju telewizji. Na terenie całego kraju zaprzestano transmisji telewizji analogowej. Zastąpiono ją telewizją cyfrową, która zapewnia lepszą jakość obrazu i dźwięku oraz dostarcza dodatkowe funkcje. Podobną drogą rozwoju podąża radio. Na terenie naszego kraju rozwija się infrastruktura cyfrowego radia.

W Polsce wykorzystywana jest technologia DAB+ (ang. *Digital Audio Broadcasting*). Jest to standard transmisji dźwięku, pozwalający na zawarcie wielu usług w jednym multipleksie. Wykorzystywana jest w nim metoda OFDM (ang. *Orthogonal Frequency-Division Multiplexing*), umożliwiająca transmisję wielu strumieni danych na ortogonalnych częstotliwościach nośnych [1].

Standard DAB+ jest rozwinięciem systemu DAB. Wykorzystuje on sprawniejszy kodek HE-AAC v2 (ang. *High Efficiency Advanced Audio Coding*) oraz lepszą korekcję błędów. Pozwoliło to na zwiększenie liczby stacji w niezmienionym paśmie.

DAB, w porównaniu do radia analogowego, niesie ze sobą takie zalety, jak:

- niższe koszty transmisji
- lepsza wydajność pasma
- lepsza jakość dźwięku
- dodatkowe funkcjonalności [2]

Technologia ta posiada również wady, takie jak:

- skomplikowana technologia
- niskie pokrycie zasięgiem na terenie Polski

W poniższej pracy podjęta zostanie próba zwiększenia możliwości zasięgowych programowalnego radia cyfrowego (ang. *SDR - Software Defined Radio*). W tym celu zostanie zaimplementowana technika SIMO (ang. *Single Input Multiple Output*) w

istniejącym już oprogramowaniu. Wykorzystane oprogramowanie pochodzi z projektu, rozwijanego przez studentów Katedry Telekomunikacji na Akademii Górniczo-Hutniczej w Krakowie.

Celem pracy jest wykorzystanie dwóch modułów antenowych w jednym odbiorniku radia programowalnego. Pozwoli to zapobiegać utracie danych w przypadku wystąpienia niektórych zakłóceń na drodze radiowej. Takie rozwiązanie może przynieść znaczne korzyści między innymi w radiach samochodowych, które mają utrudniony odbiór z powodu ciągłej zmiany pozycji względem anteny nadawczej. Technika ta mogłaby w znaczny sposób zwiększyć komfort słuchacza, ponieważ może zapobiegać zanikom dźwięku.

W pierwszym rozdziale zostanie przedstawiony opis wybranych technik radiowych. Zaprezentowane zostaną w nim podstawowe techniki oraz technologie, których poznanie jest niezbędne do zrozumienia kolejnych części pracy. Drugi rozdział pokaże jak przebiegała implementacja techniki SIMO. Analizie poddane zostaną napotkane problemy oraz zaprezentowana będzie struktura oprogramowania wraz z zastosowanymi algorytmami. W trzecim rozdziale odbędzie się analiza wyników. Przedstawione zostaną wykresy obrazujące prace techniki SIMO oraz osiągnięte efekty. Ostatnią częścią pracy jest krótkie podsumowanie, w którym zebrane będą najważniejsze wnioski .

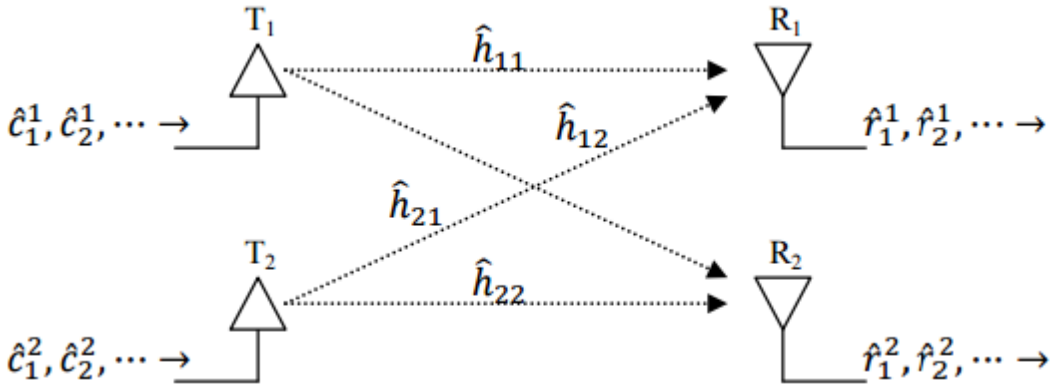
1. Opis wybranych technik radiowych

W tym rozdziale opisane zostaną głównie techniki wykorzystywane w projekcie. Zawarte są w nim szczegóły dotyczące układów odbiorczo-nadawczych wykorzystujących wiele anten (ang. MIMO – *Multiple Input Multiple Output*). Opisana zostanie technika SIMO, która została wykorzystana w projekcie. Ostatnią częścią tego rozdziału będzie wprowadzenie do standardu cyfrowego radia DAB+.

1.1 Technika MIMO

MIMO to technika umożliwiająca multipleksację przestrzenną. Realizuje, w określonym paśmie, transmisję niezależnych ciągów symboli. Wykorzystując zjawisko wielodrogowości umożliwiła zwiększenie zasięgu, bez zwiększenia ogólnej mocy emitowanej w torach nadawczych [3]. MIMO znalazło już zastosowanie w sieciach bezprzewodowych, takich jak WIFI czy sieci komórkowe 4G, a jego wykorzystanie planowane jest w sieciach komórkowych 5G [4].

Anteny nadawcze nadają w tym samym paśmie sygnały zawierające różne informacje. Sygnały zakłócają się wzajemnie, dlatego dla poprawnej detekcji wymagana jest wiedza o transmitancji kanałów radiowych.



Rys. 1.1 Schemat MIMO(2,2)

Źródło: [5]

Rysunek 1.1 przedstawia podstawową formę MIMO z dwoma antenami nadawczymi oraz dwoma antenami odbiorczymi. W powyższym układzie możemy wyróżnić cztery tory radiowe, które oznaczone zostały transmitancjami $\hat{h}_{i,j.}$. Symbol $\hat{r}$ oznacza symbol odebrany po stronie odbiorczej, natomiast $\hat{c}$ reprezentuje

transmitowany symbol. Dla przedstawionego układu możemy zapisać równanie opisujące zależności pomiędzy sygnałami dla *n-tego* symbolu:

$$\hat{r}_n^1 = \hat{h}_{11}\hat{c}_n^1 + \hat{h}_{21}\hat{c}_n^2 \qquad (1)$$

$$\hat{r}_n^2 = \hat{h}_{12}\hat{c}_n^1 + \hat{h}_{22}\hat{c}_n^2$$

Otrzymujemy w ten sposób układ dwóch równań, który dla strony odbiorczej posiada dwie niewiadome. W rozwiązaniu otrzymamy symbole, nadawane po stronie odbiorczej. Jeżeli macierz, złożona z transmitacji torów radiowych, jest nieosobliwa, to możliwe jest bezbłędne zdekodowanie informacji wysyłanych przez obie anteny. Dzięki tej zależności, w powyższym układzie, możliwe jest dwukrotne zwiększenie przepustowości w stosunku do układu SISO (ang. *Single Input Single Output*). Dla wariantu MIMO(T,R), gdzie T to liczba anten nadawczych, a R to liczba anten odbiorczych, możliwe jest zwiększenie *N* krotnie przepustowości, gdzie N to mniejsza z wartości T i R [5], [6].

Zalety techniki MIMO:

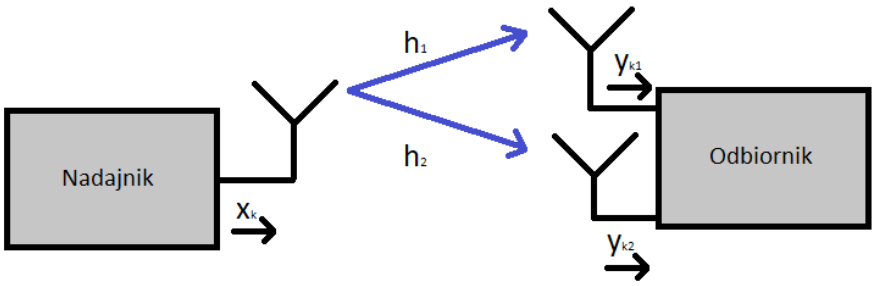
- wzrost przepływności łącza radiowego
- zwiększenie odporności na zakłócenia w torze radiowym

Wady techniki MIMO:

- komplikacja układów odbiorczych,
- zwiększona złożoność obliczeniowa

1.2 Technika SIMO

Technika SIMO jest szczególnym przypadkiem techniki MIMO. Antena nadawcza wysyła tę samą informację do wszystkich anten odbiorczych zgodnie z poniższym rysunkiem.



Rys. 1.2 Schemat układu SIMO

Dla powyższego układu można zapisać równanie dla każdej z anten odbiorczych:

$$y_{ki} = x_k h_i + n_i \qquad \textbf{(2)}$$

gdzie:

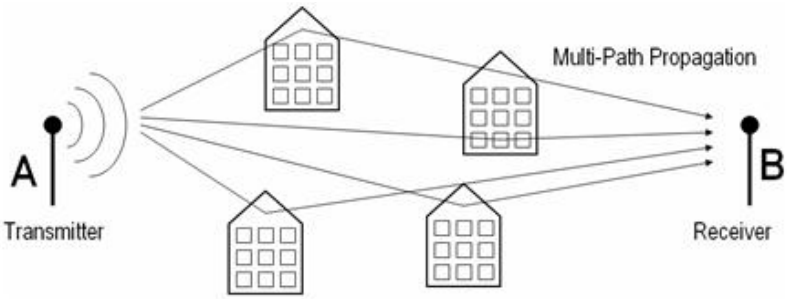
y_i - *k-ty* symbol odebrany przez *i-tą* antenę

x_k - nadany przez antenę nadawczą *k-ty* symbol

h_i - odpowiedź impulsowa *i-tego* kanału radiowego

n_i - szum w *i-tym* kanale radiowym [7]

Zjawisko wielodrogowości jest nieodłącznym elementem transmisji radiowej, a w układach wieloantenowych ma kluczowe znaczenie. Emitowany sygnał propaguje różnymi drogami (patrz rysunek 1.3), co powoduje występowanie zjawiska interferencji konstruktywnej lub dekonstruktywnej. W sytuacji niekorzystnego nałożenia się fal radiowych mamy do czynienia z głębokimi zanikami [5]. Zjawisko to szczególnie dotkliwe jest w warunkach miejskich oraz wewnątrz budynków, gdzie wielodrogowość ma szczególne znaczenie [8].



Rys. 1.3 Efekt wielodrogowości

Źródło: [12]

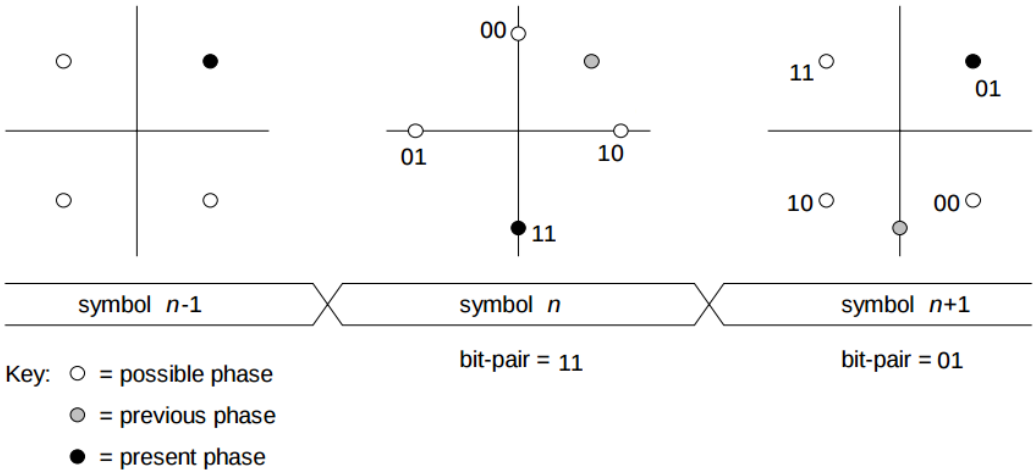
Technika SIMO nie wyeliminuje wpływu zjawiska wielodrogowości, jednak może go znacznie ograniczyć. Do anten odbiorczych docierają sygnały, które pokonały różne ścieżki. Szansa na wystąpienie zaniku destruktywnego dla wszystkich anten maleje wraz ze wzrostem ilości anten odbiorczych. Do poprawnej pracy odbiornika wystarczy, aby dla jednej z anten nie wystąpił zanik. Zadaniem odbiornika, w systemie stosującym SIMO, jest uwzględnienie otrzymanych sygnałów tych anten, z których prawdopodobnie uda się zdekodować poprawnie emitowany symbol. Jednym z najpopularniejszych algorytmów stosowanych do połączenia sygnałów ze wszystkich anten jest MRC (ang. *Maximal Ratio Combining*). Polega on na sumowaniu wszystkich

symboli, odpowiednio przemnożonych przez współczynnik wagowy [7]. W celu połączenia symboli różnych anten można zastosować też wybór sygnału o lepszych parametrach lub sumę wszystkich sygnałów.

1.3 Cyfrowe radio DAB+

Wspominany wcześniej standard DAB+ jest systemem wykorzystującym multipleksację COFDM (ang. *Coded Orthogonal Frequency Division Multiplex*). Pasmo częstotliwości podzielone jest na wiele ortogonalnych częstotliwości nośnych, na których odbywa się transmisja symboli. Podział pasma o dużej przepustowości na wiele pasm o małej przepustowości zmniejsza wpływ zjawiska wielodrogowości na system. Zawdzięcza się to wydłużeniu czasu trwania symbolu, który nie jest już porównywalny z przesunięciami pomiędzy dochodzącymi kopiami sygnału. Wykorzystywaną techniką jest również przeplot w dziedzinie czasu. Ciąg danych, docierający do nadajnika, nie jest transmitowany w całości w tym samym czasie. Daje to szanse na utrzymanie ciągłego odtwarzania przez odbiornik w wypadku zaniku trwającego pewien okres czasu. Przeplatane są cztery kolejne ramki TF (ang. *Transmission Frame*) o czasie trwania 100 ms. Przy odpowiednim poziomie parametru SNR, pojedyncza błędna ramka TF nie spowoduje utraty ciągłości dekodowania sygnału. Kolejnym wykorzystywanym usprawnieniem jest przeplot w dziedzinie częstotliwości. Polega on na nadawaniu ciągu informacji na różnych częstotliwościach podnośnych. Dzięki rozproszeniu ciągu danych podczas transmisji na szerokości całego pasma zmniejszono wrażliwość systemu na zaniki selektywne.

Standard DAB+ korzysta z modulacji π/4 DQPSK (ang. *π/4 Differential Quaternary Phase Shift Keying*), która jest zmodyfikowaną wersją modulacji QPSK (ang. *Quaternary Phase Shift Keying*). Polega ona na kodowaniu dwóch bitów poprzez zmianę fazy sygnału w sosunku do poprzednio nadawanego sygnału. Przykładowo, dla przesłania symbolu *00* sygnał jest przesuwany w fazie o 45°, natomiast dla symbolu *01* o 135° [9].



Rys. 1.4 Przebieg modulacji π/4 DQPSK.

Źródło: [9]

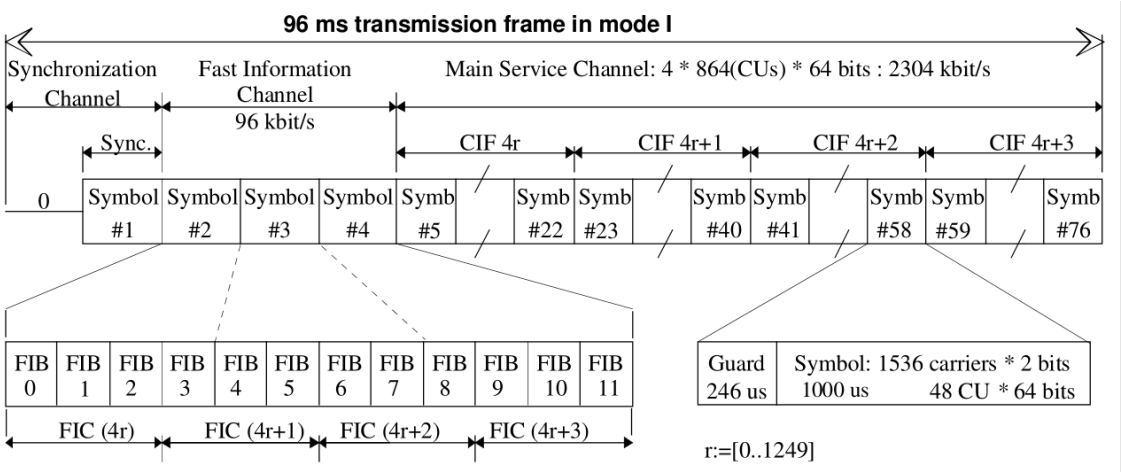
Na rysunku 1.4 przedstawiono przykładowy przebieg modulacji kolejnych symboli. Symbol *n-1* oznaczony jest czarną kropką na wykresie z lewej strony. Następny symbol, który zawiera informację *11*, przesunięty jest względem poprzedniego o -135°. Symbol *n+1*, przenoszący bity *01*, zmienił fazę o 135° względem symbolu *n-tego*. Wszystkie możliwe fazy dla danego symbolu różnią się wielokrotnością przesunięcia o 90°.

W standardzie DAB+ istnieją cztery różne tryby transmisji (ang. *Transmission Modes*). Różnią się od siebie między innymi:

- liczbą podnośnych częstotliwości
- odległością pomiędzy podnośnymi
- długością odstępu ochronnego (ang. *guard interval*)
- strukturą przesyłanych danych
- maksymalną częstotliwością pracy

Tryb transmisji dostosowywany jest do warunków w jakich będzie pracował system. Pierwszy tryb nadaje się do pokrycie dużego obszaru, ze względu na długi odstęp ochronny (ang. *guard interval*), co uodparnia system na zjawisko wielodrogowości. Wszystkie tryby transmisji korzystają z pasma częstotliwości o szerokości około 1.5MHz [2] .

Struktura ramki w standardzie DAB+ składa się z trzech kanałów, których rozmiar i budowa uzależnione są od wykorzystywanego trybu transmisji. Poniżej przedstawiona została struktura ramki dla pierwszego trybu transmisji.



Rys. 1.5 Struktura ramki w pierwszym trybie transmisji

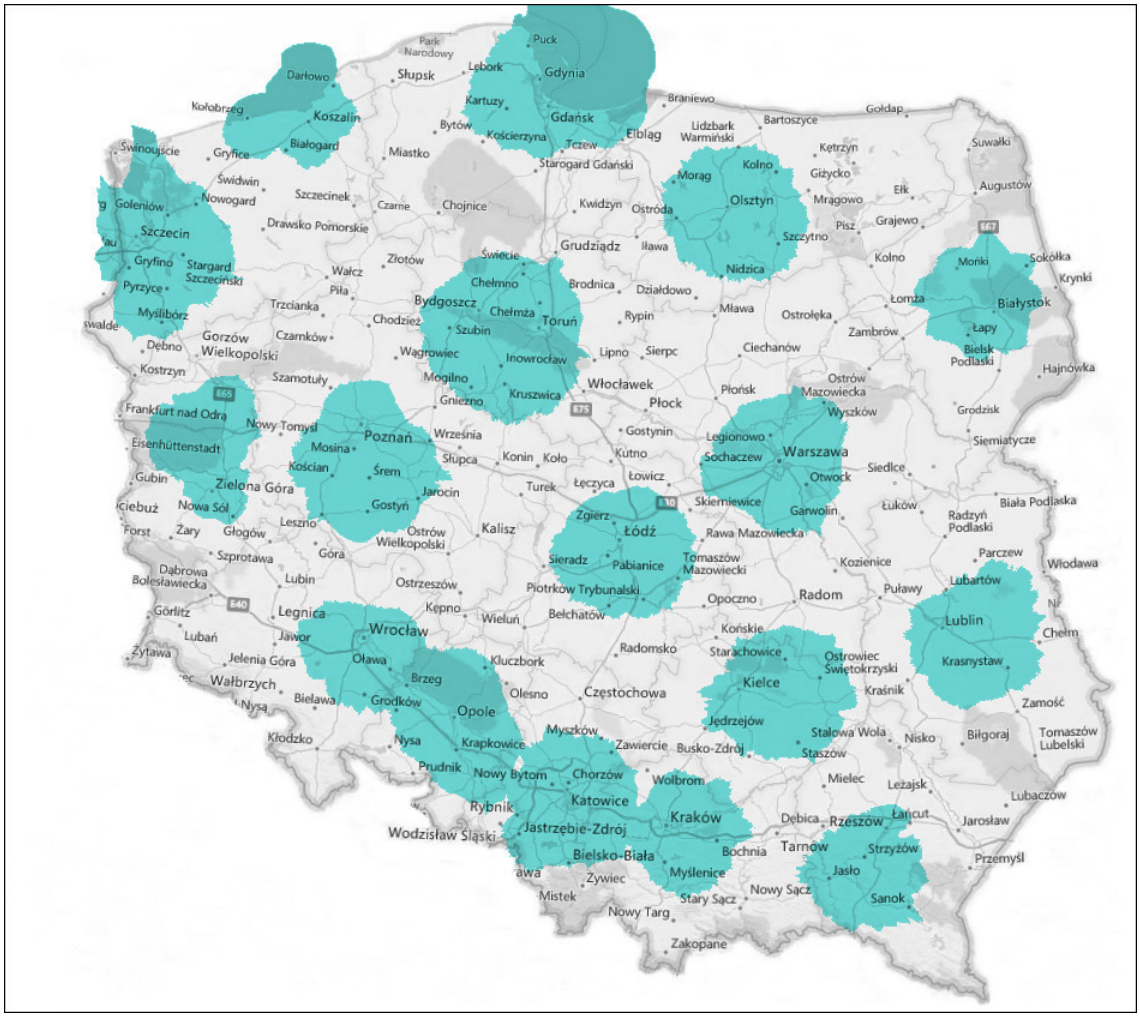
Źródło: [10]

Pierwszym z kanałów jest SC (ang. *Synchronisation Channel*) . Składa się on z symbolu *Null* oraz *symbolu Phase Reference*. Symbol *Null* służy w odbiorniku do synchronizacji początku ramki. Symbol *Phase Reference* przenosi znaną odbiornikowi informację, co pozwala na poznanie odpowiedzi impulsowej kanału radiowego [10].

Kolejnym kanałem jest FIC (ang. *Fast information channel*). Zawiera on informacje o rodzaju transmisji oraz dostępnych stacjach. Każdy FIC zbudowany jest z bloków FIB (ang. *Fast Information Block*), które składają się z elementów FIG (ang. *Fast Information Group*) przenoszących informację. Dla tego bloku nie wykorzystuje się przeplotu w dziedzinie czasu [9].

Ostatnim elementem ramki jest MSC (ang. *Main Service Channel*). W kanale tym transmitowane są przede wszystkim dane audio stacji radiowych. MSC składa się z ramek CIF (ang. *Common Interleaved Frame*), które zawierają 55 296 bity danych [10].

W naszym kraju DAB+ jest wykorzystywany w największym stopniu przez Polskie Radio, które nadaje w tym standardzie kilka różnych kanałów radiowych. Zasięgiem obejmują 54% ludności Polski, a do 2020 roku planowane jest dotarcie do wszystkich Polaków. Poniżej przedstawiona jest mapa pokrycia zasięgiem z kwietnia 2017 roku [11].



Rys. 1.6 Pokrycie zasięgiem DAB+ terenu Polski

Źródło: [11]

2. Implementacja techniki SIMO

W tym rozdziale opisana zostanie implementacja techniki SIMO. Zaprezentowane zostaną fragmenty kodu, które pomogą zobrazować wykonaną pracę oraz opisane zostanie wykorzystane w projekcie oprogramowanie.

2.1 Projekt SDR-DAB

Podczas implementacji wykorzystano istniejące już oprogramowanie radia programowalnego sdrdab, stworzone na Akademii Górniczo-Hutniczej w Krakowie w ramach przedmiotu „Radio programowalne w praktyce” [10]. Składa się na nie biblioteka *sdrdab*, interfejs linii komend *sdrdab-cli* oraz inne narzędzia , które okazały się przydatne podczas pracy nad oprogramowaniem.

Projekt został stworzony na platformy GNU/Linux, dzięki czemu dostępny jest na wielu systemach operacyjnych. Podczas implementacji wykorzystano Ubuntu 16.04 LTS.

Jednym z celi podczas tworzenia oprogramowania była wysoka wydajność. Wykorzystany został język programowania C++, który dobrze sprawdza się w takich przypadkach. Daje on programiście dobrą kontrolę nad pamięcią operacyjną. Ponadto, umożliwia programowanie zorientowane obiektowo (ang. *object-oriented programing*), które oferuje:

- elastyczność – łatwo modyfikowalny kod
- przyswajalność – szybsze zrozumienie struktury programu
- wielokrotne wykorzystanie tego samego kodu

Radio daje możliwość wyboru jednego z dwóch typów źródeł danych. Pierwszym jest tuner DVB-T posiadający układ scalony RTL2832U, który połączony jest z platformą poprzez wejście USB. Drugą możliwością są dane zapisane w pliku. Oba źródła dostarczają do radia próbki z kanału kwadraturowego oraz kanału synfazowego, każda o wielkości 8 bitów. Przy częstotliwości próbkowania 2.048 kHz otrzymujemy strumień danych o szybkości 32.768 Mbps, który należy przetworzyć w czasie rzeczywistym.

Jednym z elementów projektu jest interfejs linii komend. Pozwala on między innymi na zmianę stacji w czasie działania programu oraz wypisanie wszystkich

dostępnych stacji radiowych. Innym przydatnym narzędziem jest *sdrtool*, które pozwala na:

- budowanie projektu w różnych wariantach
- instalacja środowiska uruchomieniowego
- uruchamianie testów
- pobieranie dodatkowych elementów
- generowanie dokumentacji [10]

2.2 Struktura programu

Sdrdab w znacznym stopniu polega na mechanizmach wielowątkowości. Wymagane jest to aby utrzymać ciągłość w dekodowaniu sygnału i odtwarzania dźwięku.

Występują w nim wątki:

- DATAFEEDER
- RESAMPLER
- SYNCHRONIZER
- DEMODULATOR
- DATADECODER
- AUDIO

Wątek *DATAFEEDER* odpowiedzialny jest za dostarczanie danych do programu, poprzez zapisywanie ich do odpowiedniego bufora. Źródłem danych może być plik lub moduł USB. W rozwiązaniu tego problemu posłużono się mechanizmem polimorfizmu z języka C++, co pozwoliło na stworzenie ujednoliconego interfejsu. Zastosowano abstrakcyjną klasę *AbstractDataFeeder* po której dziedziczą klasy *FileDataFeeder* oraz *RtlDataFeeder*. Implementują one różne sposoby pozyskiwania danych, ale w kodzie mogą być wykorzystywane wymiennie. Użycie tych klas przedstawiono na rysunku 2.1.

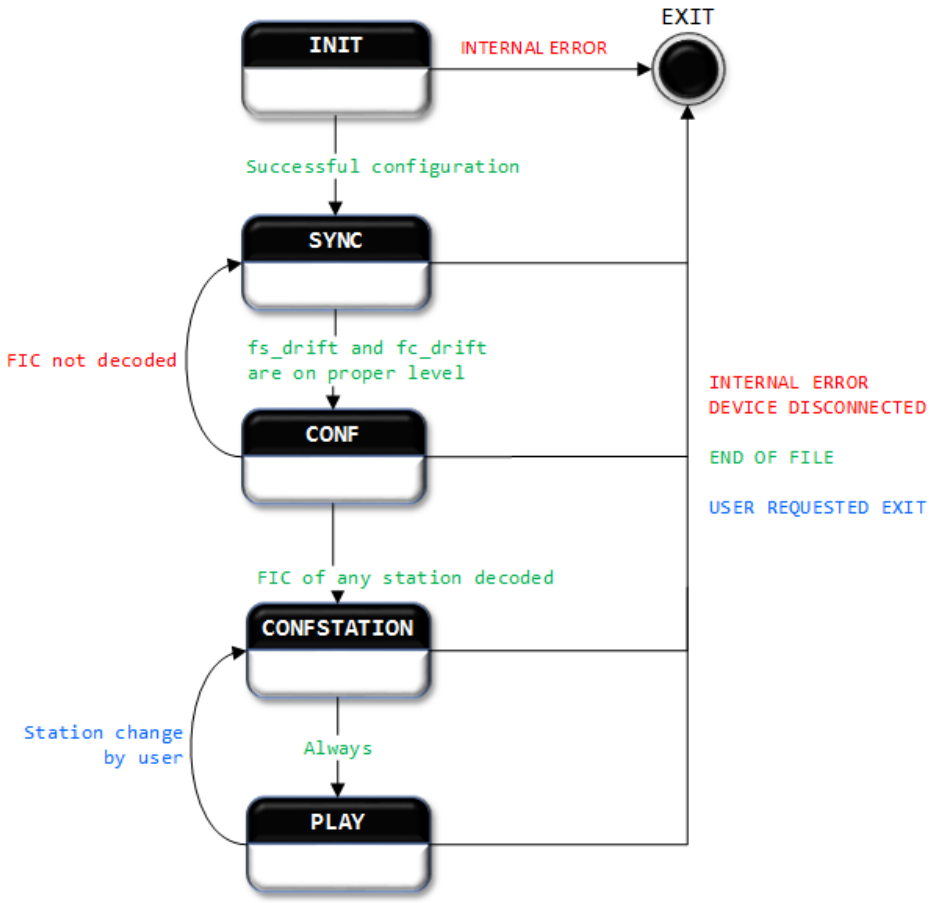
```
if ( data_source == DATA_FROM_FILE ) {
    datafeeder_ = new FileDataFeeder( dongle_or_file_name,
                                      internal_buffer_size,
                                      sample_rate,
                                      carrier_freq,
                                      10 );
} else if ( data_source == DATA_FROM_DONGLE ) {
    datafeeder_ = new RtlDataFeeder( dongle_or_file_name,
                                     internal_buffer_number,
                                     internal_buffer_size,
                                     sample_rate,
                                     carrier_freq,
                                     10 );
} else {
    fprintf( stderr, "DataFeeder object creation fail, returning..." );
    return INTERNAL_ERROR;
}
```

Rys. 2.1 Tworzenie obiektów klas pochodnych, dziedziczących po klasie *AbstractDataFeeder*

Klasa *Scheduler* odpowiedzialna jest za poprawne działanie całego programu. Jej zadaniem jest utworzenie wątków oraz rezerwowanie i zwalnianie pamięć buforów, z których korzystają poszczególne moduły systemu. W celu organizacji pracy, została ona wyposażona w maszynę stanów, która jest zrealizowana w formie instrukcji wyboru *switch* (rysunek 2.2). Schemat jej został przedstawiony na rysunku 2.3. Stany zostały zdefiniowane jako typ wyliczeniowy (ang. *enumerate*) [10].

```
switch ( state_ ) {
case SYNC:
    state_ = Sync();
    break;
case CONF:
    state_ = Conf();
    break;
case CONFSTATION:
    state_ = Reconfigure();
    break;
case CONFCONVALG:
    state_ = Reconfigure();
    break;
case PLAY:
    state_ = Play();
    break;
case EXTERNAL_STOP:
    program_end_delay_ = 2;
    if (verbose_)
        fprintf(stderr, "External stop has been requested. \n");
    break;
case INTERNAL_ERROR:
    program_end_delay_ = 2;
    if (verbose_)
        fprintf(stderr, "Error in Process function: internal error occurs. \n");
    break;
default:
    if (verbose_)
        fprintf( stderr, "Error in Process function: machine state enters default case. \n" );
}
```

Rys. 2.2 Implementacja tzw. maszyny stanów



Rys. 2.3 Schemat maszyny stanów

Źródło: [10]

Proces uruchamiania radia rozpoczyna się od stanu *INIT*, w którym incjalizowane są zmienne. Dodatkowo, jest on odpowiedzialny za tworzenie wątków, do czego wykorzystuje klasę *SignaledWorkerThread*. Podczas jej inicjalizacji uruchamiany jest nowy wątek. Jego zadaniem jest wykonanie, przekazanej przez argument metody, a następnie oczekiwanie na ponowne wznowienie. Wątki wybudzane są przez obiekt klasy *Scheduler*, który wywołuje na obiekcie klasy *SignaledWorkerThread* funkcję *resume_thread*. Instancje klasy *SignaledWorkerThread* przechowywane są w tabalicy *threads_*, *a* identyfikowane są poprzez zdefiniowany typ wyliczeniowy *thread_ids_t*. Wątek po zakończeniu określonego działania zwraca do klasy *Scheduler* informację o wykonanej pracy. W tym celu wykorzystywana jest jest klasa *BlockingQueue*, która tworzona jest w klasie zarządzającej, a następnie przekazywana jest do wątków podczas inicjalizacji. Spełnia ona rolę kolejki (ang. *queue*), struktury typu FIFO (ang. *First In First Out*). Ponadto, posiada zaimplementowane mechanizmy współdzielenia pomiędzy wątkami. Jest to konieczne, aby uniknąć próby dostępu do wspólnych danych w tym

samym czasie. Poniżej (rysunek 2.4) przedstawiona została implementacja metody *push*, która jest podstawowym elementem struktury typu kolejka.

```
void push(T element) {
    ScopedLock slock(lock_);
    queue_.push(element);
    pthread_cond_signal(&cond_);
}
```

Rys 2.4 Metoda *push* klasy *BlockingQueue*

Następnym etapem w maszynie stanów jest *SYNC*. Odpowiada on za zlokalizowanie początku symbolu *NULL* oraz detekcję trybu transmisji. W tym stanie wznawiany jest wątek *SYNCHRONIZER*. Jego zadaniem jest synchronizacja czasowa oraz częstotliwościowa. Znajduje on początek ramki, a następnie oblicza przesunięcie częstotliwości nośnej, które przykładowo może być spowodowane nagrzaniem się modułu antenowego. Dokładne zlokalizowanie symbolu *NULL* ma szczególne znaczenie dla kolejnych etapów synchronizacji. Podczas tworzenia obiektu klasy Synchronizer wyznaczane są współczynniki symbolu referencyjnego, które służą nam do ustalenia pozycji symbolu *Phase Reference* [13].

Kolejnym stanem jest *CONF*. W tym etapie dekodowany jest kanał FIC. Pozwala to na zebranie informacji o stacjach nadawanych przez odbiornik. W strukturze *stationInfo* zostaje umieszczona lista stacji radiowych dostępnych w multipleksie wraz z parametrami potrzebnymi do ich dekodowania.

Następnym stanem jest *CONFSTATION*. Dokonuje on konfiguracji systemu dekodowania, aby odpowiadała wybranej z listy stacji.

Ostatni stan to *PLAY*. Dekodowany jest w nim FIC oraz MSC. Jego zadaniem jest podtrzymywanie ciągłej transmisji dźwięku. W tym celu obsługuje kolejkę *BlockingQueue*, którą uzupełniają wątki po zakończeniu pracy. Wykorzystuje do tego instrukcję wyboru *switch*.

2.3 Implementacja

Program postanowiono dostosować do techniki SIMO i przetestować z dwiema antenami odbiorczymi, jednak zmiany przeprowadzane były tak, by możliwa była łatwa rozbudowa odbiornika o kolejne anteny. Implementacja SIMO wymagała dużych zmian kodu źródłowego oryginalnego dekodera, głównie w klasie *Scheduler*, która

nadzoruje poprawność działania radia. Zdecydowano się na wydzielenie części kodu i stworzenie nowej klasy *Source*. Obiekty tej klasy tworzone są przez przez klasę *Scheduler*, która przetrzymuje je w strukturze danych typu *vector<Source *>* o nazwie *sources* (rysunek 2.5).

```
for(int i=0;i<numOfSources;i++){
    sources.push_back(new Source());
    sources[i]->sourceId=i;
}
```

Rys. 2.5 Tworzenie obiektów klasy *Source*

Klasa jest odpowiedzialna za:

- dostarczanie danych,
- synchronizację.

Stan *SYNC* jest teraz wykonywany niezależnie dla każdego obiektu *Source*. Jest to konieczne, ponieważ mamy do czynienia z niezależnymi modułami odbiorników radiowych. Elementy tych układów mają różne parametry fizyczne, w inny sposób starzeją się oraz nagrzewają, co ma wpływ na ich działanie. Z tego powodu dla każdego ze źródeł jest konieczny osobny proces kompensacji. Zmiany maszyny stanów, dostosowujące ją do nowych warunków zostały przedstawione na rysunku 2.6.

```
switch ( state_ ) {
case SYNC:
    for(int i=0;i<numOfSources;i++){
        do{
            state_ = sources[i]->Sync();
        }while(state_!=CONF);
    }
    break;

case CONF:
    state_ = Conf();
    break;

case CONFSTATION:
    state_ = Reconfigure();
    break;

case CONFCONVALG:
    state_ = Reconfigure();
    break;

case PLAY:
    state_ = Play();
    break;
```

Rys. 2.6 Zmieniona maszyna stanów

Dekoder oczekuje, aż wszystkie źródła danych zsynchronizują się, dopiero wtedy zostanie rozpoczęta procedura przejścia do stanu CONF.

Zmiana ta spowodowała utworzenie kolejnych wątków, które związane są z dostarczaniem danych oraz synchronizacją.

Wymagało to rozszerzenia istniejącego typu wyliczeniowego *thread_ids_t*, który odpowiedzialny jest za identyfikację wątków. Na rysunku 2.7 przedstawiona została stosowana deklaracja typu *enum*.

```
enum thread_ids_t {
    DATAFEEDER_PROCESS_THREAD_ID_0 = 0,
    DATAFEEDER_PROCESS_THREAD_ID_1,
    RESAMPLE_THREAD_ID_0,
    RESAMPLE_THREAD_ID_1,
    AUDIODECODER_PROCESS_THREAD_ID,
    SYNCHRONIZER_PROCESS_THREAD_ID_0,
    SYNCHRONIZER_PROCESS_THREAD_ID_1,
    CALCULATE_SNR_THREAD_ID,
    DEMODULATOR_PROCESS_THREAD_ID,
    DATADECODER_PROCESS_THREAD_ID,
    AUDIO_PROCESS_THREAD_ID,
    NUMBER_OF_THREADS
};
```

Rys. 2.7 Deklaracja typu wyliczeniowego *thread_ids_t*

Każdy obiekt klasy *Source* posiada wątki:

- DATAFEEDER
- RESAMPLER
- SYNCHRONIZER

Wymagało to wydzielenia części metody *CreateThreads* klasy *Scheduler* do metody *CreateSourceThreads* klasy *Source*. Odbywa się w niej inicjalizacja wątków dla poszczególnych źródeł sygnału. Metoda *CreateThreads* wywołuje *CreateSourceThreads* dla każdego obiektu, co przedstawiono na rysunku 2.8.

```
for(std::vector<Source *>::iterator source = sources.begin();source!=sources.end();source++){
    if (  (*source)->CreateSourceThreads(threads_, &threads_event_queue_) == INTERNAL_ERROR ) {
        return INTERNAL_ERROR;
    }
}
```

Rys. 2.8 Inicjalizacja wątków obiektu *Source*

Wszystkie wątki korzystają z jednego obiektu klasy *BlockingQueue*, który służy do raportowania po wykonanym zadaniu.

Stan *CONF* oczekuje, aż źródła osiągną synchronizację, co umożliwi demodulację oraz zdekodowanie kanału FIC. W tym celu wykorzystuje klasy *Demodulator* oraz *DataDecoder*. Podczas wykonywania zadań związanych z demodulacją dokonywane jest łączenie źródeł pochodzących z niezależnych anten.

Stan *PLAY* również uległ modyfikacji. Jest on odpowiedzialny za obsługę kolejki *BlockingQueue*, więc wymagana była implementacja obsługi nowych wartości z typu wyliczeniowego *thread_ids_t*. Obsługa realizowana jest poprzez instrukcje warunkową *switch*, co ułatwiło rozbudowę algorytmu o kolejne warunki. Przykładowo obsługa wątku *DATAFEEDER* po modyfikacji pozwalającej obsługiwać 2 obiekty klasy *Source* (patrz rysunek 2.9).

```
case DATAFEEDER_PROCESS_THREAD_ID_0:

    // If DataFeeder stored data, lock mutex, set pointer to end of wrote data.
    if (sources[0]->data_write_.data_stored) {
        sources[0]->last_data_feeder_write_ = sources[0]->data_write_pos_ +
                                              sources[0]->data_write_.block_size;
        sources[0]->frame_time_=sources[0]->data_write_.time;
    }

    sources[0]->ShiftSyncFeederPointersIfPossible(sources[0]->demod_read_pos_queue_,
                                                  sources[0]->demod_read_pos_);
    sources[0]->ResumeSynchronizerIfReady();
    break;
case DATAFEEDER_PROCESS_THREAD_ID_1:
    // If DataFeeder stored data, lock mutex, set pointer to end of wrote data.
    if (sources[1]->data_write_.data_stored) {
        sources[1]->last_data_feeder_write_ = sources[1]->data_write_pos_ +
                                              sources[1]->data_write_.block_size;
        sources[1]->frame_time_=sources[1]->data_write_.time;
    }

    sources[1]->ShiftSyncFeederPointersIfPossible(sources[1]->demod_read_pos_queue_,
                                                  sources[1]->demod_read_pos_);
    sources[1]->ResumeSynchronizerIfReady();
    break;
```

Rys. 2.9 Obsługa wątków *DATAFEEDER*

Obok klasy *Scheduler* największej modyfikacji uległa klasa *Demodulator*, która wykorzystywana jest przez wątek *DEMODULATOR.* W niej zostało zaimplementowane łączenie buforów z niezależnych źródeł.

Wykorzystywane, przez obiekt klasy *Demodulator*, bufory zostały powielone o liczbę obiektów *Source* (rysunek 2.10). Dodane zostały również nowe bufory, które były niezbędne w procesie łączenia danych (rysunek 2.10).

```
numberOfLastReferenceSymbols_=20;
ofdm_symbol_ = new float **[numberOfSources_];
sync_start_data_ =new float *[numberOfSources_];
output_snr_ =new float *[numberOfSources_];
deqpsk_ =new float*[numberOfSources_];
ofdm_symbol_reference_signal_=new float**[numberOfSources_];
for(int i=0;i<numberOfSources_;i++)
{
    ofdm_symbol_reference_signal_[i]=new float*[mode_parameters_->number_of_carriers];
    ofdm_symbol_[i] = new float*[mode_parameters_->number_of_symbols];
    sync_start_data_[i] = new float[mode_parameters_->null_size];
    output_snr_[i] = new float[mode_parameters_->number_of_symbols];
    deqpsk_[i] = new float[76*2 * mode_parameters_->number_of_carriers];
}
for(int i=0;i< mode_parameters_->number_of_carriers;i++)
{
    for(int j=0;j<numberOfSources_;j++){
        ofdm_symbol_reference_signal_[j][i]=new float[numberOfLastReferenceSymbols_];
        for(int r=0;r<numberOfLastReferenceSymbols_;r++){
            ofdm_symbol_reference_signal_[j][i][r]=-1.0;
        }
    }
}
reference symbol index =0;
```

Rys. 2.10 Alokacja buforów klasy *Demodulator*

Do wątku *DEMODULATOR* przekazywany jest jako argument obiekt *demodReadWrite*. Jest on odpowiedzialny za przekazanie informacji o tym gdzie w buforze znajdują się przetwarzane dane oraz w jakim miejscu mają zostać zapisane zdemodulowane dane. Wskaźnik, przenoszący informację o miejscu gdzie znajdują się dane, został zastąpiony wektorem, który zawiera informację o poszczególnych obiektach *Source*.

Demodulator dla każdego ze źródeł wykonuje sekwencję wywołań funkcji (rysunek 2.11). *CalculateFramePosition* oblicza początki symboli OFDM oraz zapisuje je do tablicy *ofdm_symbol_*. Następnie obliczany jest SNR dla każdego ze źródeł oraz przeprowadzana jest transformacja FFT (ang. *Fast Fourier Transform*). Ostatnią funkcją wywoływaną dla każdego ze źródeł jest *DeQPSK*. Jest ona odpowiedzialna za demodulację sygnału. Wszystkie powyższe funkcję zmodyfikowane zostały tak, by mogły być wywoływane dla różnych buforów.

```
for(int i=0;i<data_input_output->read_here_vec.size();i++)
{
    if(data_input_output->read_here_vec[i].first)
    {
        if (symb_end_addr_ == mode_parameters_->number_symbols_per_cif - 1)
            memcpy(sync_start_data_[i],
                   data_input_output->read_here_vec[i].second +
                   2 * mode_parameters_->frame_size - 3 * mode_parameters_->null_size,
                   sizeof(float) * mode_parameters_->null_size);

        CalculateFramePosition(data_input_output->read_here_vec[i].second,i);
        SNRcalc(output_snr_[i],i);
        FFTInPlace(mode_parameters_->fft_size,i);
        DeQPSK(deqpsk_[i],i);
    }
}
MergeDeQPSK(data_input_output->write_here);
```

Rys. 2.11 Proces domodulacji sygnału.

Następnym krokiem, po przeprowadzeniu demodulacji, jest połączenie wszystkich buforów oraz zapisanie ich do pamięci wskazywanej przez wskaźnik *write_here*, co jest realizowane funkcją *MergeBuffers(...)*.

Funkcja ta posiada pięć różnych sposobów łączenia buforów, które identyfikowane są poprzez zadeklarowany typ *mergeSignalOption* (rysunek 2.12)

```
enum mergeSignalOption {
    SCALE_BY_SNR = 0,
    CHOOSE_SIGNAL_WITH_BETTER_SNR,
    CHOOSE_SOURCE_ID1,
    CHOOSE_SOURCE_ID2,
    ADD_SIGNALS
};
```

Rys. 2.12 Deklaracja typu *mergeSignalOption*

CHOOSE_SOURCE_ID1 oraz *CHOOSE_SOURCE_ID2* pozwalają korzystać z tylko jednego źródła danych. Dane zostaną skopiowane z jednego bufora do drugiego bez żadnej zmiany.

ADD_SIGNALS to opcja służąca do dodawania symboli. Jej działanie polega na sumowaniu części rzeczywistych oraz części urojonych pochodzących z różnych odbiorników antenowych.

CHOOSE_SIGNAL_WITH_BETTER_SNR zapisuje do bufora wynikowego dane ze źródła, który ma lepszy stosunek sygnału do szumu. Wybór pomiędzy źródłami, ze względu na wartość SNR, dokonywany jest dla każdej częstotliwości podnośnej osobno.

SCALE_BY_SNR służy do skalowania wartości pochodzących z różnych źródeł. Dodawane są do siebie symbole, które przemnożone są przez odpowiednie wagi. Ich znaczenie zależy od wartości SNR poszczególnych częstotliwości podnośnych.

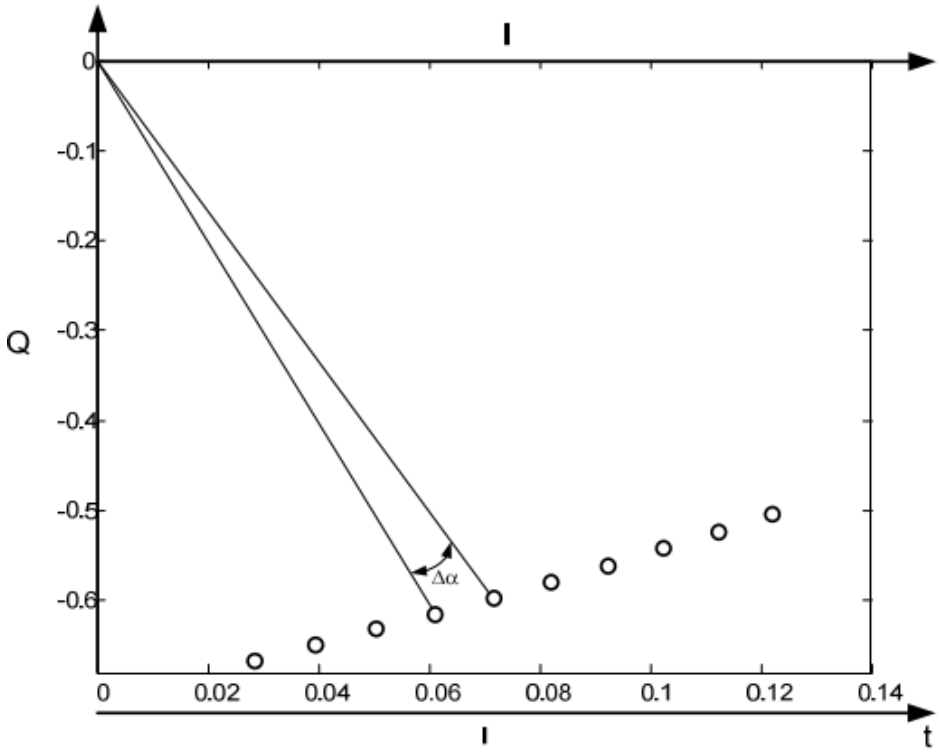
```
switch(SNRoptionChoice_){
    case SCALE_BY_SNR:{
        float SNRsum=0;
        for(int src = 0 ;src < numberOfSources_;src++){
            SNRsum += snr[src];
        }
        *write_real = 0;
        *write_imag = 0;
        for(int src = 0 ;src < numberOfSources_;src++){
            *write_real += (*real[src])*snr[src]/SNRsum;
            *write_imag += (*imag[src])*snr[src]/SNRsum;
        }
        break;}
    case ADD_SIGNALS:{
        *write_real = 0;
        *write_imag = 0;
        for(int src = 0 ;src < numberOfSources_;src++){
            *write_real += *real[src];
            *write_imag += *imag[src];
        }
        break;}
    case CHOOSE_SOURCE_ID1:{
        *write_real = *real[0];
        *write_imag = *imag[0];
        break;}
    case CHOOSE_SOURCE_ID2:{
        *write_real = *real[1];
        *write_imag = *imag[1];
        break;}
    case CHOOSE_SIGNAL_WITH_BETTER_SNR:{
        int maxSNR=0;
        for(int src = 0 ;src < numberOfSources_;src++){
            if(snr[src]>snr[maxSNR])maxSNR=src;
        }
            *write_real = *real[maxSNR];
            *write_imag = *imag[maxSNR];
        break;}
}
```

Rys. 2.13 Implementacja łączenia buforów

Na rysunku 2.13 przedstawiono implementację łączenia buforów. Zrealizowana została ona za pomocą instrukcji warunkowej *switch*. W zależności od wartości zmiennej *SNRoptionChoice*, wybierane są odpowiednie sposoby scalania danych. Zmienne *write_real* oraz *write_imag* są wskaźnikami typu *float**. Wskazują na obszar pamięci, który zostanie później przekazany do dekodera. Tablice *real* oraz *imag* zawierają wskaźniki do części rzeczywistych i urojonych wszystkich źródeł sygnału.

CHOOSE_SIGNAL_WITH_BETTER_SNR oraz *SCALE_BY_SNR* do poprawnego działania wymagają znajomości parametru SNR. Zastosowana modulacja DQPSK moduluje sygnał poprzez zmianę fazy w stosunku do poprzednio przesyłanego symbolu. Uodparnia to transmisję na błędy fazy oraz pozwala na detekcje niekoherentną, czyli bez potrzeby posiadania sygnału odniesienia [14]. Jednak podczas transmisji może wystąpić efekt „obracania się” konstelacji (rysunek 2.14). Jest on

spowodowany różnicą częstotliwości odbieranego sygnału oraz heterodyny demodulatora [15].



Rys. 2.14 Efekt obracającej się konstelacji zaobserwowany podczas nadawania sygnału bez zmiany fazy

Źródło: [15]

Korzystając ze znajomości cech modulacji DQPSK oraz standardu DAB postanowiono na obliczenie wartości SNR z użyciem symbolu *Phase Reference*. Przenosi on informację, która jest znana po stronie odbiorczej. Pozwala to na zastosowanie go jako sygnału odniesienia. Do obliczenia wartości SNR posłużono się zależnością:

$$SNR = \frac{SymPow}{|SymPow - RefSymPow|} \quad \textbf{(3)}$$

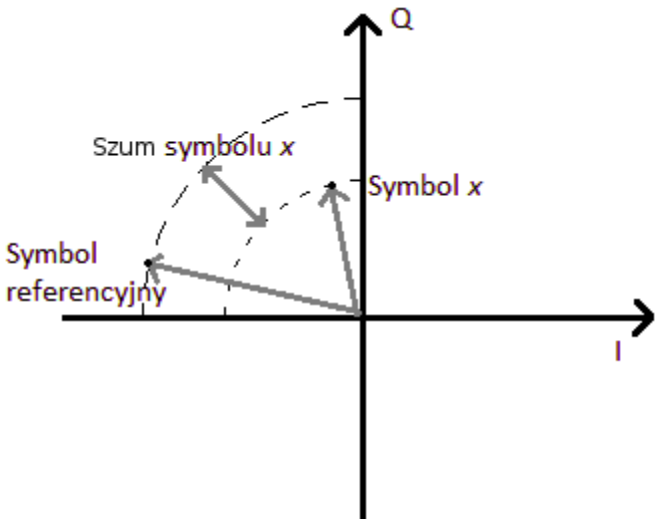
gdzie:

SymPow – moc sygnału symbolu, dla którego obliczamy SNR

RefSymPow – moc sygnału symbolu referencyjnego (*Phase Reference)*

Wykorzystuje ona wartości mocy, które wyliczane są jako moduły liczb zespolonych. Metoda została tak zrealizowana, by wykorzystać cechy stosowanej modulacji. DQPSK

kompensuje błędy fazowe, więc nie musimy ich uwzględniać podczas obliczania wartości SNR. Cecha ta sprawia, że nasze obliczenia możemy oprzeć na wartościach mocy. Jako moc szumu symbolu *x-tego* przyjmujemy bezwzględną różnicę mocy symbolu referencyjnego oraz mocy symbolu *x-tego*. Moc symboli referencyjnych (*Phase Reference*) oraz wartości SNR są obliczane niezależnie dla każdej z częstotliwości podnośnych dla każdego ze źródeł danych. Metodę tę można zobrazować na poniższej konstelacji (rysunek 2.15).



Rys. 2.15 Metoda obliczania SNR zobrazowana na konstelacji

W kodzie zastosowano uśrednianie mocy symboli referencyjnych. Parametr *numberOfLastReferenceSymbols_* określa z ilu ostatnich ramek wartości mają zostać uwzględnione. Są one przetrzymywane dla wszystkich źródeł oraz podnośnych w tablicy *ofdm_symbol_reference_signal_*.

```
snr[src]=sqrt(ofdm_symbol_[src][i][ind] *
              ofdm_symbol_[src][i][ind] +
              ofdm_symbol_[src][i][ind+1] *
              ofdm_symbol_[src][i][ind+1]);

if(i==2)
{
    float referencePower =sqrt(ofdm_symbol_[src][0][ind] *
                               ofdm_symbol_[src][0][ind] +
                               ofdm_symbol_[src][0][ind+1] *
                               ofdm_symbol_[src][0][ind+1]);
    ofdm_symbol_reference_signal_[src][j/2][reference_symbol_index_] =
                                                          referencePower;
}
float noise=0;
int r;
for(r=0;r<numberOfLastReferenceSymbols_;r++){
    if(ofdm_symbol_reference_signal_[src][j/2][r]<0)
        break;
    noise+=ofdm_symbol_reference_signal_[src][j/2][r];
}
if(r>0){
    noise/=float(r);
}
else{
    noise=50;
}
snr[src]/=std::abs(snr[src]-noise);
```

Rys. 2.16 Implementacja obliczania wartości SNR

Tablica *snr* służy do przetrzymywania wartości SNR dla wszystkich źródeł, które później są wykorzystywane podczas łączenia buforów. Zmienna lokalna *noise* jest używana do obliczenia średniej mocy z ostatnich *numberOfLastReferenceSymbols_* symboli referencyjnych.

3. Analiza wyników

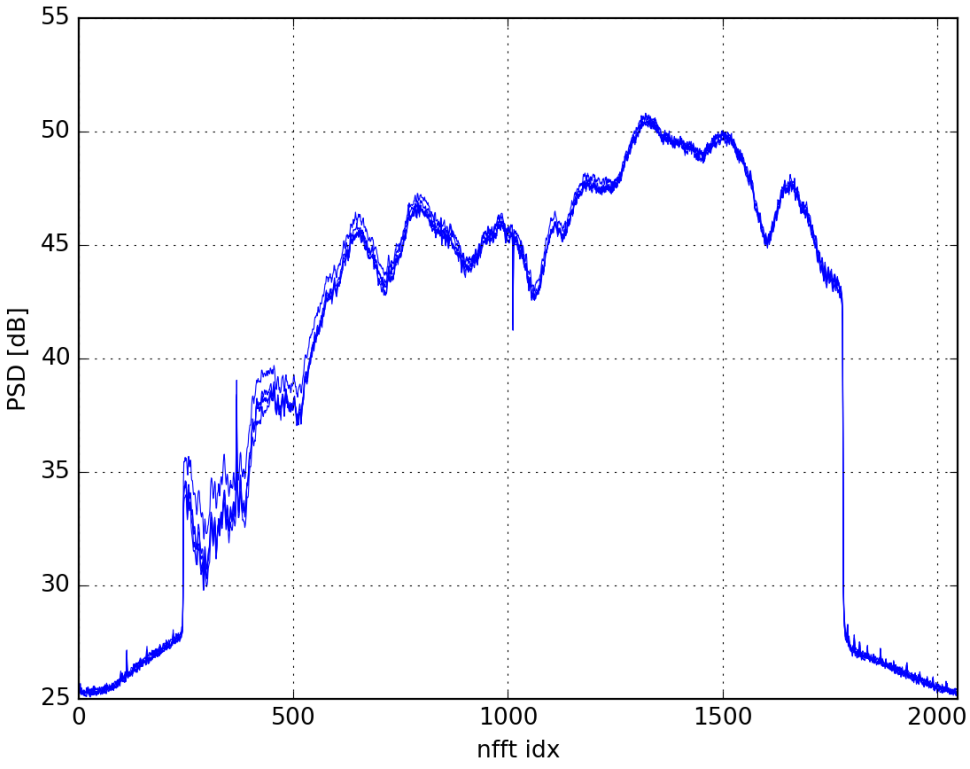
Ostatnim etapem pracy jest analiza osiągniętych wyników. Aby była ona wiarygodna, postanowiono badać układ za pomocą plików zawierających próbki sygnału. Podczas trwania programu mierzona była ilość zdekodowanych bajtów audio.

Pierwsze próbki, które zostały poddane analizie, zostały zebrane na terenie Krakowa. Sygnał nadawany był z Radio-Telewizyjnego Centrum Nadawczego w pobliżu Wieliczki. Wykorzystuje on pierwszy tryb nadawana standardu DAB+ [10]. Do zebrania danych posłużyła komenda wywołana pod systemem operacyjnym Ubuntu (patrz rysunek 3.1).

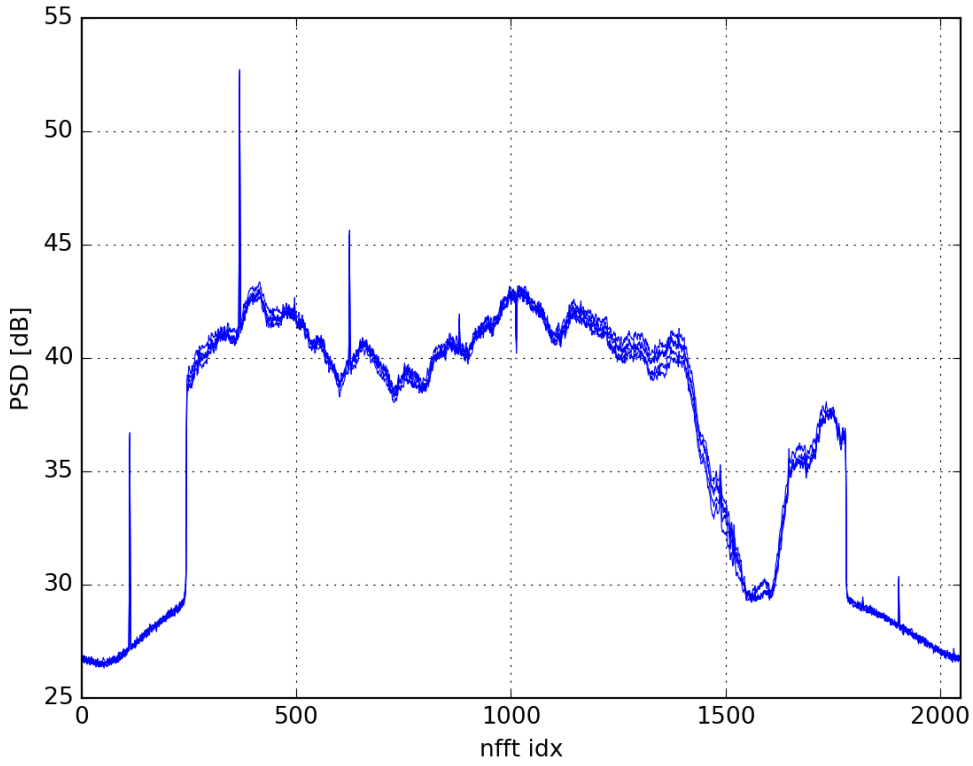
```
michal@michal-S551LN:~$ rtl_sdr  -f 218640000 -g 28 -s 2048000 -n 50000000  -d 1
 antena1 & rtl_sdr  -f 218640000 -g 21 -s 2048000 -n 50000000  -d 0 antena2
```

Rys. 3.1 Komenda do rejestracji danych z anten

W celu analizy widma sygnału został wykorzystany skrypt napisany w języku Python. Dla zebranych próbek otrzymano następujące wykresy:

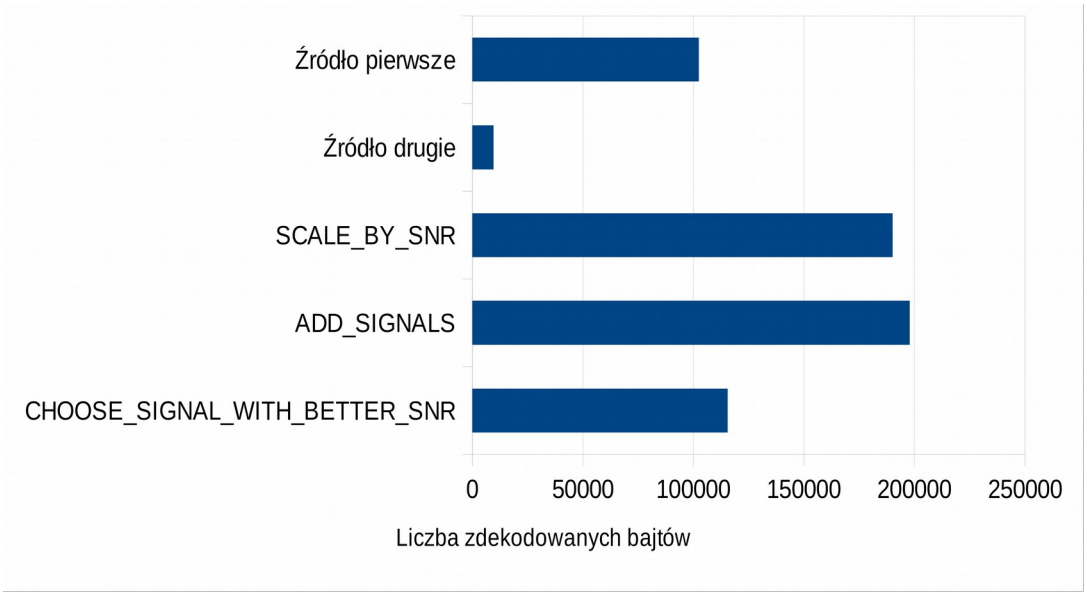


Rys. 3.2 Widmo sygnału źródła pierwszego



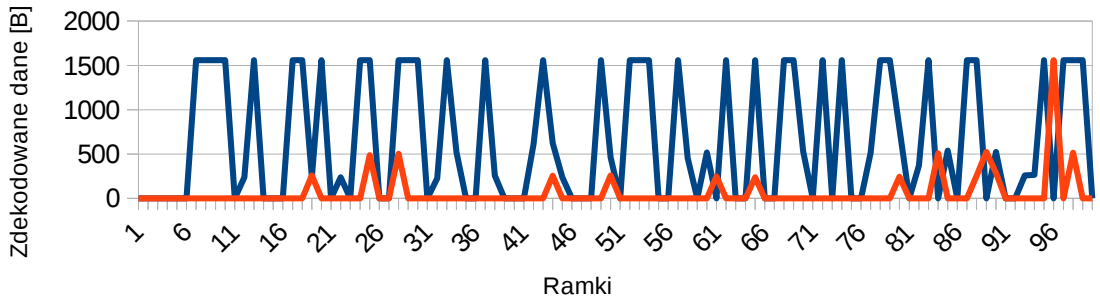
Rys. 3.3 Widmo sygnału źródła drugiego

Kolejnym etapem było sprawdzenie ilości danych audio zdekodowanych przez radio. W tym celu został zaimplementowany licznik bajtów danych audio, które zostały odtworzone. Testowanie pojedynczych plików danych odbywało się na wersji bazowej oprogramowania, na której podjęto się implementacji techniki SIMO.

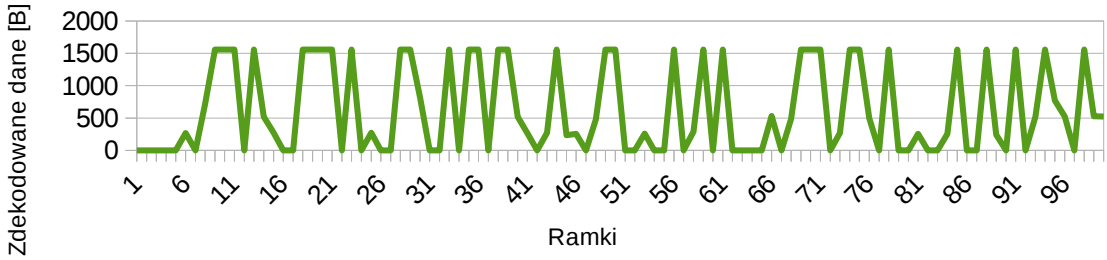


Rys. 3.4 Porównanie ilości odebranych danych

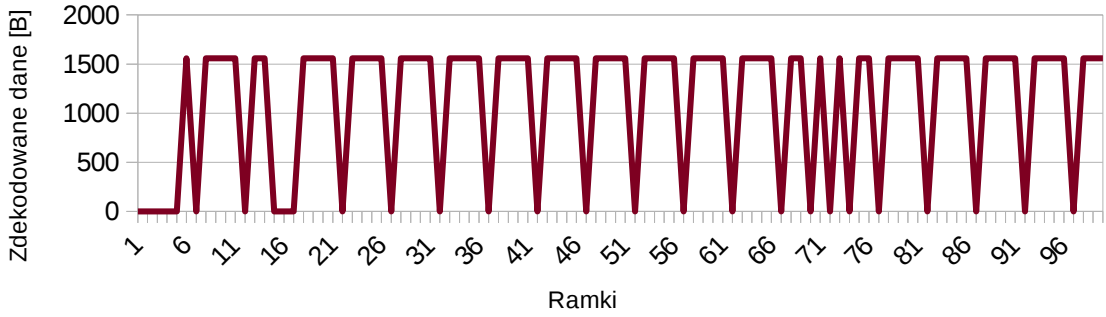
Najefektywniejszą metodą połączenia źródeł sygnału okazał się sposób *ADD_SIGNALS*. Udało się dzięki niej zdekodować prawie dwa razy więcej użytecznych danych. Podobną skuteczność miał *SCALE_BY_SNR*. *CHOOSE_SIGNAL_WITH_BETTER_SNR* uzyskał około 10% lepszy wynik od najlepszego pojedynczego źródła.



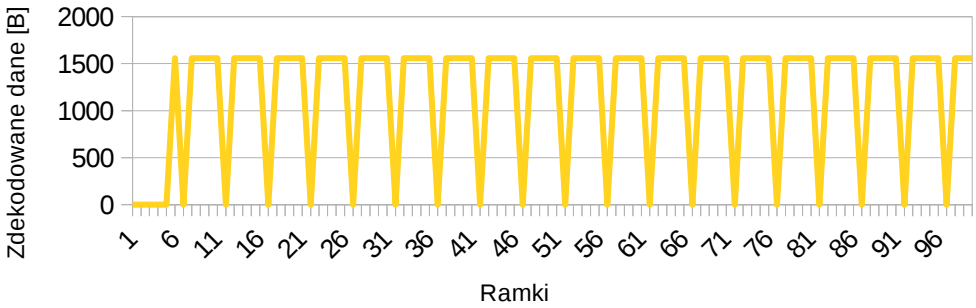
Rys. 3.5 Ilość zdekodowanych danych z kolejnych ramek dla pojedynczych źródeł



Rys. 3.6 Ilość zdekodowanych danych z kolejnych ramek dla metody *CHOOSE_SIGNAL_WITH_BETTER_SNR*



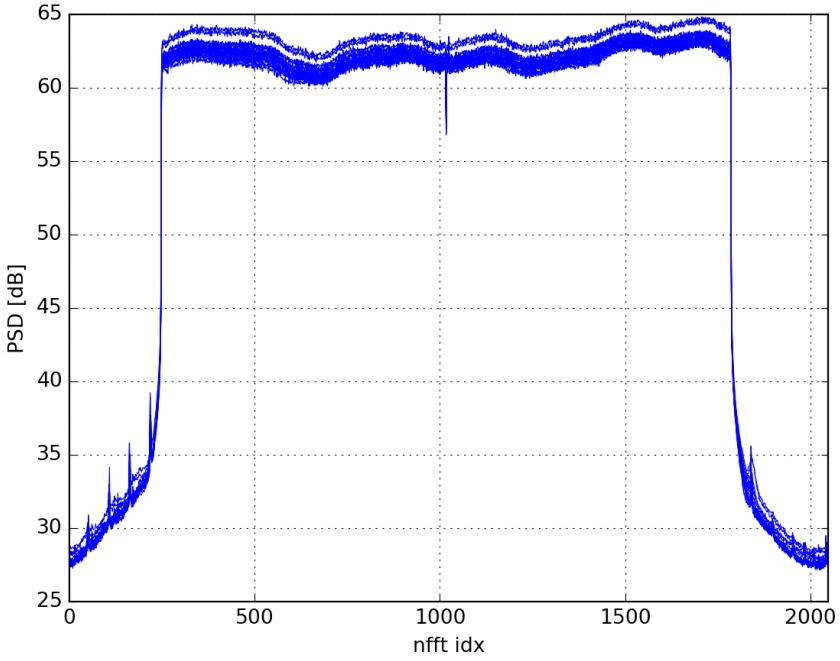
Rys. 3.7 Ilość zdekodowanych danych z kolejnych ramek dla metody *SCALE_BY_SNR*



Rys. 3.8 Ilość zdekodowanych danych z kolejnych ramek dla metody *ADD_SIGNALS*

Rysunki 3.5, 3.6, 3.7 oraz 3.8 przedstawiają liczbę zdekodowanych bajtów danych audio z kolejnych ramek. W badanym trybie transmisji na każdych pięć kolejnych ramek przypadają cztery ramki zawierające dane audio, co jest spowodowane sposobem tworzenia ramek audio (ang. *Super Frame)*. Dla dekodowanej stacji każda ramka zawiera 1559 B danych. Mniejsza liczba zdekodowanych danych oznacza uszkodzenie ramki. Medota *CHOOSE_SIGNAL_WITH_BETTER_SNR* osiągnęła nieznaczną poprawę w stosunku do sygnału pojedynczego. Natomiast dla pozostałych dwóch technik zauważalna jest regularność w liczbie dekodowanych bajtów. Dla *SCALE_BY_SNR* zaobserwować możemy zaniki dekodowanych danych, które wystąpiły dla 4 ramek. W przypadku metody *ADD_SIGNALS* nie zaobserwowano podobnych zaników.

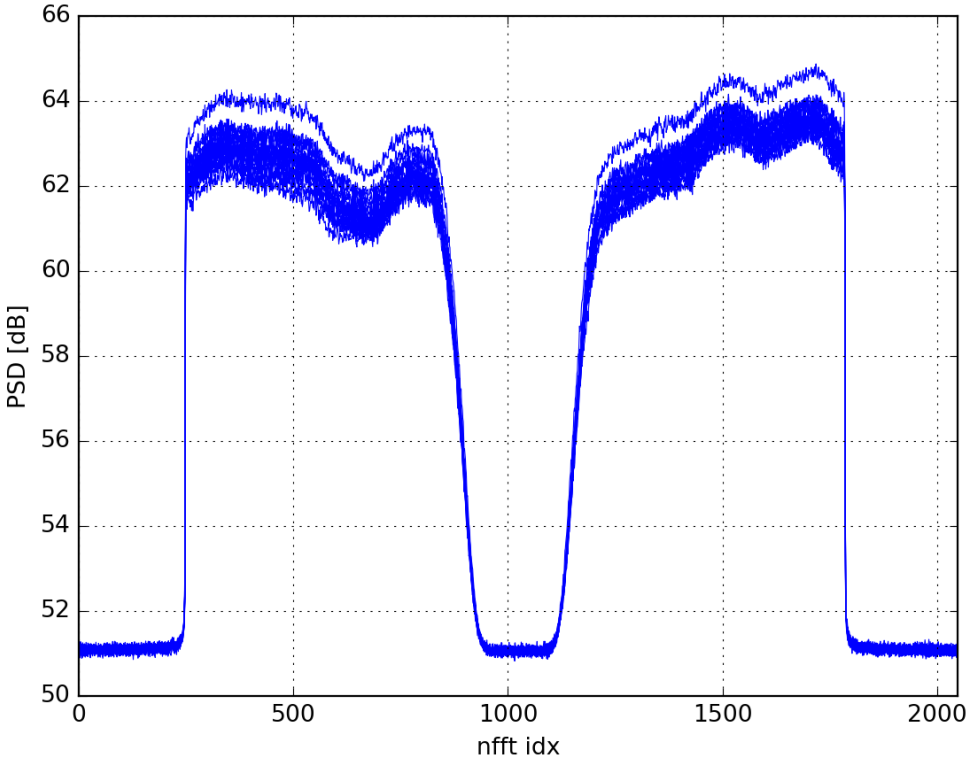
W następnym etapie analizy oprogramowania wykorzystano plik z próbkami *antena-1_dab_229072kHz_fs2048kHz_gain42_1_long.raw*, który zamieszczony jest na oficjalnej stronie projektu uczelnianego *sdrdab* [10]. Poniżej przedstawione jest jego widmo (rysunek 3.9).



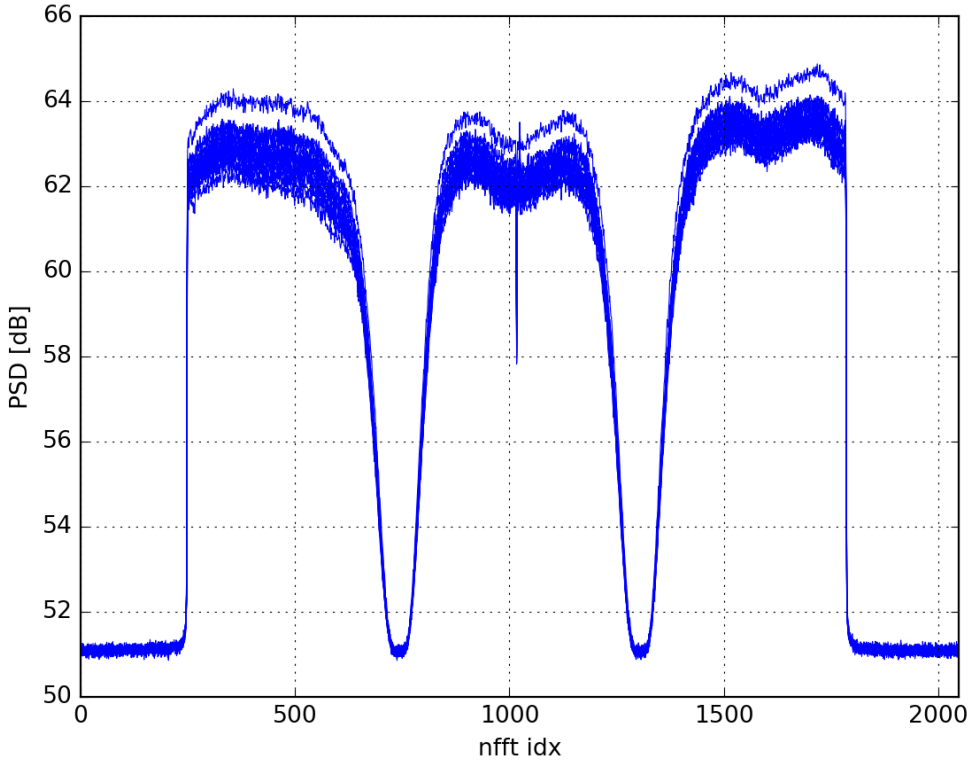
Rys. 3.9 Widmo sygnału pochodzącego ze strony projektu

Charakteryzuje się on dobrą jakością i jest łatwo dekodowalny przez radio. Został on poddany modyfikacji, by umożliwić generację kontrolowanego zaniku selektywnego. W tym celu został napisany skrypt w języku Python. Wykorzystuje on filtr Remeza z modułu *scpiy.signal* oraz funkcję *np.random.normal()* do generowania liczb losowych o rozkładzie Gaussa w celu zaszumienia sygnału.

Modyfikacji nie poddano sygnału od samego początku, w celu ułatwienia początkowej synchronizacji. Na rysunkach 3.10 i 3.11 przedstawiono widma zmodyfikowanych sygnałów.

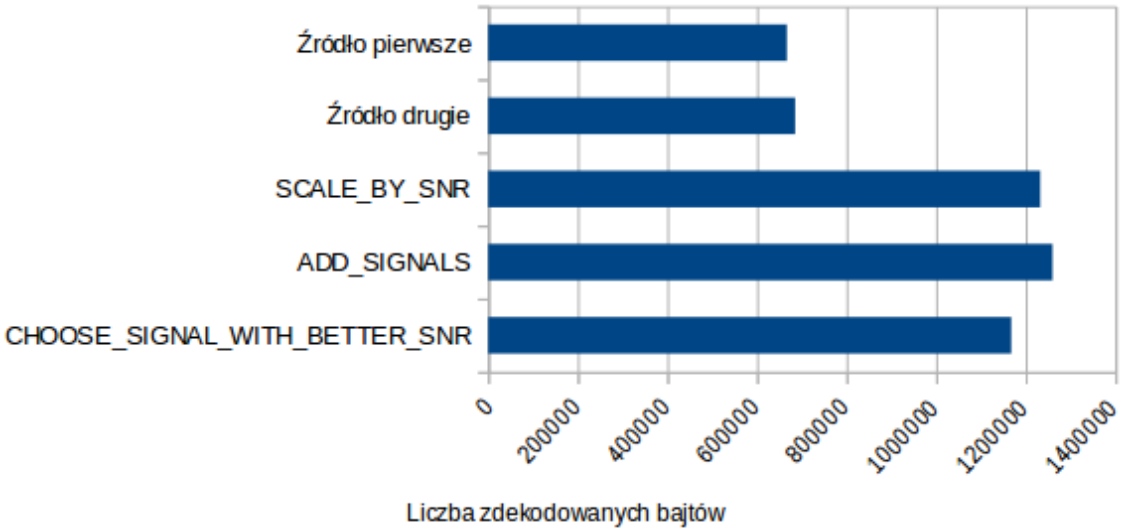


Rys. 3.10 Widmo zmodyfikowanego sygnału dla źródła pierwszego



Rys. 3.11 Widmo zmodyfikowanego sygnału dla źródła drugiego

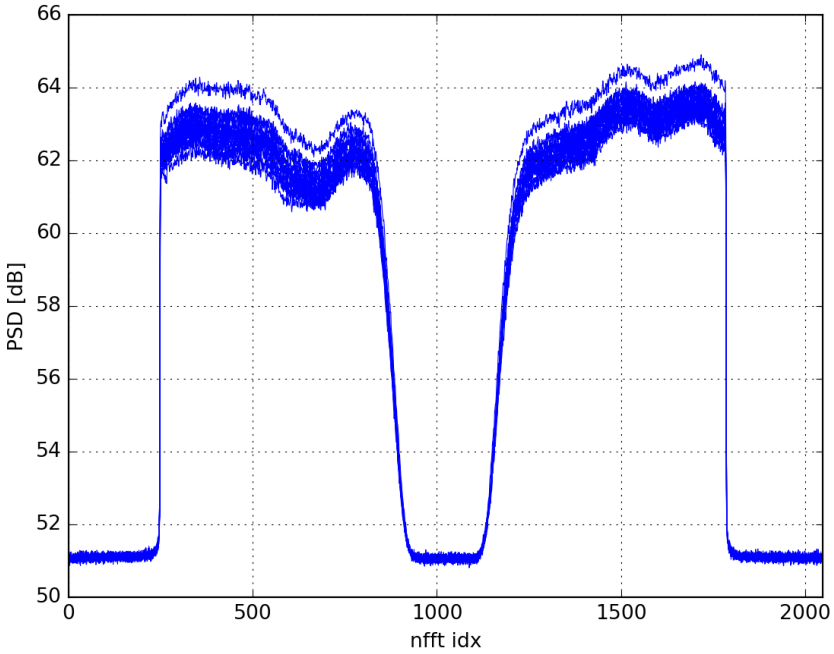
Tak zmodyfikowane sygnały poddano ponownej analizie jak w poprzednim przypadku. Sprawdzono ilość zdekodowanych bajtów danych dla pojedynczych źródeł oraz dla trzech różnych metod ich łączenia. Poniżej przedstawione są wyniki przeprowadzonej analizy (rysunek 3.12).



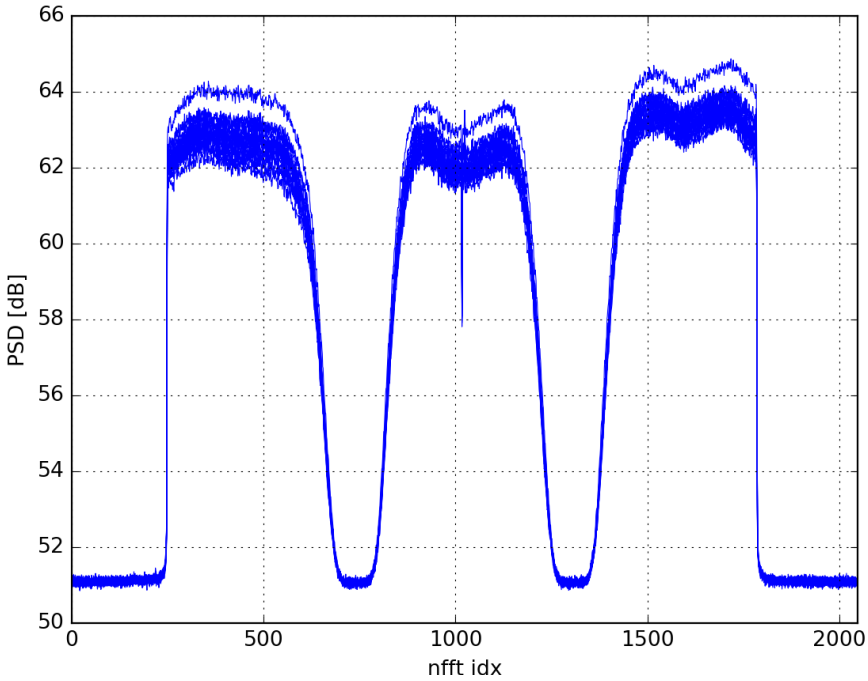
Rys. 3.12 Porównanie odebranej ilości danych dla zmodulowanych sygnałów

Dla tego przykładu zostały osiągnięte porównywalne wyniki dla wszystkich trzech metod łączenia źródeł danych. Każde zdekodowało poprawnie około dwukrotnie więcej danych niż którekolwiek z pojedynczych. Dla słuchacza oznaczała to znaczącą poprawę jakości odbieranego dźwięku, poprzez upłynnienie transmisji.

Następnie postanowiono zwiększyć pasmo zaporowe w filtrze modyfikującym próbki. Pozwoliło to na sprawdzenie zachowania oprogramowania dla dużych zaników selektywnych.

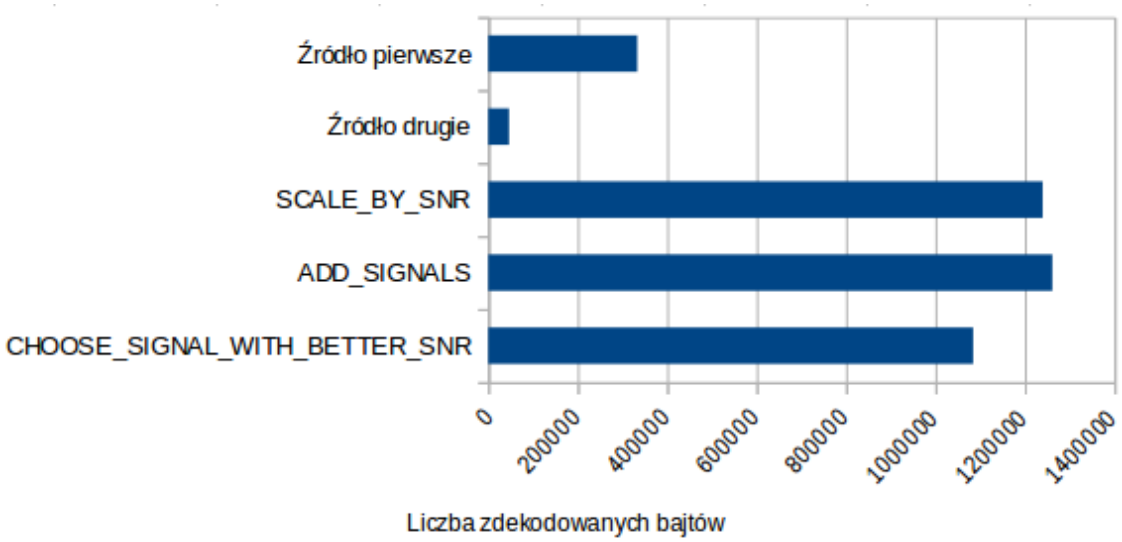


Rys. 3.13 Zmodyfikowane widmo dla źródła pierwszego



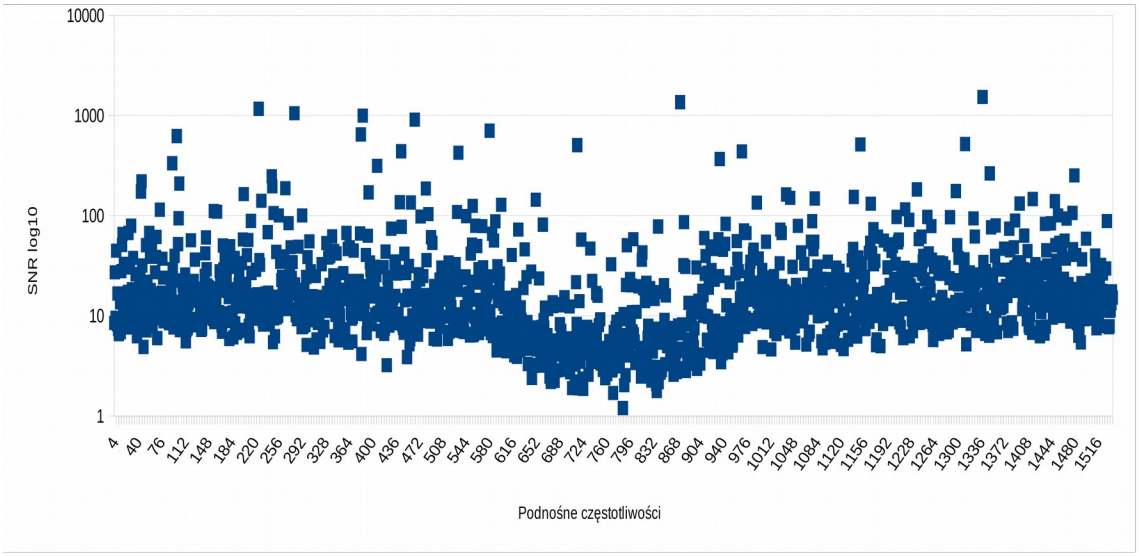
Rys. 3.14 Zmodyfikowane widmo dla źródła drugiego

Dla tak zmodyfikowanych sygnałów przeprowadzono ponownie badania dekodera.

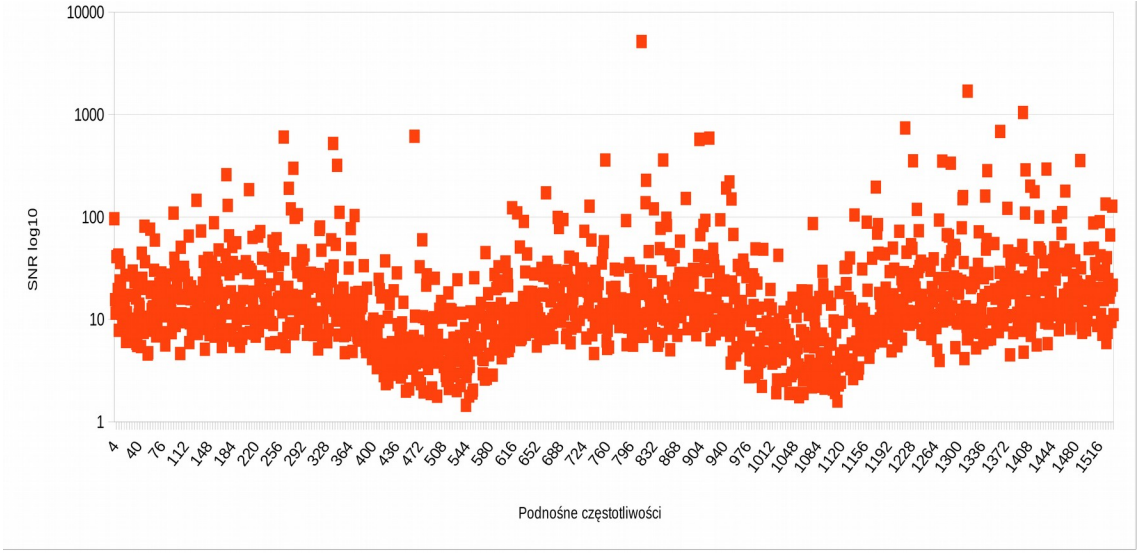


Rys. 3.15 Porównanie odebranej ilości danych dla zmodulowanych sygnałów

W powyższym przypadku źródło drugie okazało się być zmodulowane na tyle, że ilość odtworzonego audio była praktycznie zerowa. Pierwsze źródło dostarczało dźwięku który był często przerywany. Dla tego zestawu danych również najskuteczniejszą metodą była *ADD_SIGNALS*. Pozwoliła ona na zdekodowanie około trzykrotnie większej ilości danych w stosunku do techniki SISO. *SCALE_BY_SNR* okazała się porównywalnie dobre. Nieco gorzej wypadła metoda *CHOOSE_SIGNAL_WITH_BETTER_SNR*, której udało się zdekodować o około 10% danych mniej. Przewaga metody *ADD_SIGNALS* nad *SCALE_BY_SNR* oraz *CHOOSE_SIGNAL_WITH_BETTER_SNR* może być spowodowana niedokładnością pomiaru współczynnika SNR, który ma kluczowe znaczenie dla tych metod.



Rys. 3.16 Średnie wartości SNR dla poszczególnych podnośnych częstotliwości sygnału pierwszego



Rys. 3.17 Średnie wartości SNR dla poszczególnych podnośnych częstotliwości sygnału drugiego

Na rysunkach 3.16 oraz 3.17 zaobserwować możemy uśrednione wartości SNR z dziesięciu ramek dla wszystkich podnośnych częstotliwości w skali logartmicznej. Rysunek 3.16 przedstawia wartości dla sygnału o widmie z rysunku 3.13, natomiast rysunek 3.17 dotyczy sygnału o widmie z rysunku 3.14. W miejscach występowania zaników selektywnych zaobserwować można zmniejszanie się wartości SNR dla danego źródła. Zależność ta nie okazała się jendak na tyle znacząca, aby osiągnąć lepsze efekty niż w przypadku metody *ADD_SIGNALS*.

Podsumowanie

Wraz z rozprzestrzenianiem się standardu DAB+ rośnie potrzeba rozwoju odbiorników radiowych. Dla uzyskania satysfakcji słuchacza wymagana jest płynność transmitowanego dźwięku, na którą znaczny wpływ ma jakość odbieranego sygnału. Warunek ten jest szczególnie trudny do zapewnienia, w przypadku odbiorników radiowych, które zmieniają pozycję względem anteny nadawczej. Przede wszystkim radia zamontowane w samochodach mogą mieć trudność z płynną transmisją dźwięku. Technika SIMO może okazać się kluczowa dla tego standardu. Zaprezentowane rozwiązanie znacząco poprawiło działanie odbiornika. Udało się osiągnąć znaczący wzrost ilości zdekodowanych danych. W praktyce, takie rozwiązanie może pozwolić na utrzymanie płynności pracy podczas jazdy samochodem. Zastosowanie może znaleźć również w radiach stacjonarnych, które zlokalizowane są w miejscach gdzie sygnał jest mocno zniekształcony.

Podsumowując, rezultatem pracy jest zaimplementowana technika SIMO w odbiorniku cyfrowym radia DAB+. Udało się osiągnąć około dwukrotny wzrost prawidłowo zdekodowanych danych, co można uznać za satysfakcjonujący rezultat. Największych zmian w kodzie dokonano podczas modyfikacji klasy *Scheduler* oraz tworzeniu klasy *Source*. Zmiany te dokonywane były w plikach *scheduler.cc* oraz *scheduler.h* i dotyczyły około 60% ich zawartości, czyli ponad 1600 linii kodu. Podczas implementacji łączenia buforów zmienionych oraz dodanych zostało łącznie około 600 linii kodu.

W przyszłości, oprogramowanie to można znacząco rozwinąć. Jednym z elementów, który wymaga dopracowania, jest sposób obliczania wartości SNR. Dekoder można także rozwinąć o obsługę kolejnych anten. Zmiany implementacyjne prowadzone były tak, by było to w przyszłości ułatwione. Ponadto, kod wymaga optymalizacji, która umożliwi jego działanie na większej ilości platform.

Bibliografia

[1] ETSI: *Digital Audio Broadcasting; Our Role and Activities* ,Adres: http://www.etsi.org/technologies-clusters/technologies/broadcast/dab [Data uzyskania dostępu: 13.12.2017r.]

[2] W. Hoeg i T. Lauterbach: *Digital Audio Broadcasting: Principles and Applications of DAB, DAB + and DMB, Third Edition*, Wiley, 2009

[3] Yevhen Yashchyshyn, Sebastian Kozłowski, Anna Łysiuk: *Nowe techniki transmisji radiowej; Laboratorium*, Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 2015

[4] Erik G. Larsson i Liesbet Van der Perre: *Massive MIMO for 5G*, Adres: https://5g.ieee.org/tech-focus/march-2017/massive-mimo-for-5g [Data uzyskania dostępu:13.12.2017r.]

[5] Jacek Gruca: Rozprawa doktorska: *Analiza xastosowania techniki MIMO w bezprzewodowych sieciach sensorycznych*, Kraków 2012

[6] Branka Vucetic, Jinhong Yuan: *Space-Time Coding*, Wiley 2003

[7] Adam Lipka: *Badanie metod transmisji w systemach wieloantenowych – MIMO*;Praca nr 08300028,Gdańsk, grudzień 2008

[8] Paweł Kułakowski: Praca magisterska: *Analiza wpływu warunków terenowo-klimatycznych na pracę systemów radiokomunikacyjnych*, Kraków 2003

[9] C. Gandy: *DAB: an introduction to the Eureka DAB System and a guide to how it works*, Technical Report WHP-061, British Broadcasting Corp, 2003.

[10] Dokumentacja biblioteki sdrdab, Adres: https://sdr.kt.agh.edu.pl/sdrdab [Data uzyskania dostępu: 15.12.2017r.]

[11] Polskie Radio: Cyfrowe Radio DAB+; Zasięg, Adres: https://www.polskieradio.pl/240,Cyfrowe-radio-DAB/4698,zasieg [Data uzyskania dostępu: 18.12.2017r.]

[12] National Instruments: *The Perfect Simulation" for Wireless Receiver Test* ,Adres: http://www.ni.com/white-paper/6427/en/ [Data uzyskania dostępu: 18.12.2017r.]

[13] Sebastian Leśniak: Praca inżynierska: *Estymacja i korekcja przesunięcia częstotliwości nośnej w radiu cyfrowym DAB/SDR*, Kraków 2016

[14] Krystyna Maria NOGA,:Zeszyty Naukowe Wydziału Elektrotechniki i Automatyki Politechniki Gdańskiej Nr 36: *Modulacje analogowe i cyfrowe w środowisku Mathcad i Vissim*, Adres: , Gdynia, 2013

[15] Ryszard Studański, Michał Brewka, Agnieszka Studańska, Radosław Wąs: Zeszyty naukowe akademii marynarki wojennej rok lii nr 4 (187): C*yfrowy odbiór sygnałów systemu Inmarsat M*, 2011

Załącznik A

Spis zawartości dołączonej płyty CD

- Michał_Rzepka_SIMO.doc - tekst pracy zapisany w formacie MS Word
- Michał_Rzepka_SIMO.pdf - tekst pracy zapisany w formacie PDF
- DAB – katalog projektowy zawierający dekodera
- samples0, samples1 – próbki samodzielnie zarejestrowanego sygnału, na którym były przeprowadzane badania

Komendy przydatne podczas uruchamiania programu:

./sdrtool install-deps – instalacja środowiska

./sdrtool build Debug - buildowanie oprogramowania w wersji debugowalej

./build/bin/sdrdab-cli -v –open-file=samples – uruchomienie radia