POLITECHNIKA GDAŃSKA | WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

## Dokumentacja Projektu grupowego

# Dokumentacja techniczna projektu

**Wydział Elektroniki, Telekomunikacji i Informatyki**
**Politechnika Gdańska**

**{wersja dokumentu wzorcowego: wersja 2/2023}**

| | | |
|---|---|---|
| **Nazwa i akronim projektu:**<br>{nazwa projektu, np: System zabezpieczenia portu przed zagrożeniami terrorystycznymi - SZP}<br>**Aplikacja wizualizująca zagadnienia związane z kodowaniem nadmiarowym** | **Zleceniodawca:**<br>{nazwa/nazwisko klienta}<br>**dr inż. Bartosz Czaplewski** | |
| **Numer zlecenia:**<br>{numer zespołu projektowego w ramach Projektu grupowego wg systemu SPG, np. 13@KSSR'2022}<br>**5@KSTI'2023/24** | **Kierownik projektu:**<br>{kierownik zespołu projektowego}<br>**Bartosz Kołakowski** | **Opiekun projektu:**<br>{opiekun projektu}<br>**dr inż. Bartosz Czaplewski** |

| | |
|---|---|
| **Nazwa / kod dokumentu:**<br>**Dokumentacja techniczna produktu – DTP** | **Nr wersji:**<br>{wersja dokumentu np. 1.00}<br>**5.00** |
| **Odpowiedzialny za dokument:**<br>{nazwisko, imię}<br>**Kołakowski Bartosz** | **Data pierwszego sporządzenia:**<br>{data wykonania pierwszej wersji dokumentu}<br>**09.01.2024** |
| | **Data ostatniej aktualizacji:**<br>{data wykonania aktualnej wersji dokumentu}<br>**08.06.2024** |
| | **Semestr realizacji Projektu grupowego:**<br>{wpisać 1 lub 2}<br>2 |

**Historia dokumentu**

| Wersja | Opis modyfikacji | Rozdział / strona | Autor modyfikacji | Data |
|---|---|---|---|---|
| 1.00 | {opis np. wstępna wersja}<br>Wersja wstępna | {np. całość}<br>całość | {nazwisko, imię}<br>Kołakowski Bartosz | {data zmiany}<br>09.01.2024 |
| 2.00 | {opis np. poprawka w p.2.2}<br>Zmiana 5. rysunku i 25. rysunku | {np. pkt 2.2}<br>2.2.1, 2.2.5 | {nazwisko, imię}<br>Kołakowski Bartosz | {data zmiany}<br>09.01.2024 |
| 3.00 | Aktualizacja opisu ogólnego, dodanie opisu kodu Reeda-Solomona | 2.1, 2.3 | Jastrzębski Paweł | 22.05.2024 |
| 4.00 | Opis generowania exe | 2.1 | Jastrzębski Paweł | 29.05.2024 |
| 5.00 | Przeredagowanie rozdziału 2.2. Dodanie rozdziału 2.2.4 opisującego okna przy wyborze algorytmu Reeda-Solomona. Dodanie rozdziału 2.5 opisującego tłumaczenie aplikacji. | 2.2 i jego podrozdziały, 2.5 | Noga Piotr | 08.06.2024 |

***{UWAGA: w II semestrze dokumentacja może być rozszerzeniem dokumentacji z semestru I (nowa wersja dokumentu), może być też nowym plikiem;***

***UWAGA: Jeżeli dokumentacja została wytworzona za pomocą innego narzędzia, to należy wskazać plik z dokumentacją w niniejszym dokumencie jako załącznik do tego dokumentu**; tworzenie dokumentacji w inny sposób nie zwalania od obowiązku wytworzenia niniejszego dokumentu, ale zamiast opisu dokumentacyjnego wystarczy wskazać na dokument wytworzony w inny sposób}*

# Spis treści



## 1 Wprowadzenie - o dokumencie

### 1.1 Cel dokumentu

{nie zmieniać} Celem dokumentu jest udokumentowanie informacji dotyczących produktu, jego cech funkcjonalnych, parametrów technicznych, schematów blokowych, oprogramowania,

wyników działania, zdjęć produktu, pomiarów, testów oraz innych elementów wymaganych przez opiekuna i klienta.

### 1.2 Zakres dokumentu

{określenie, co wchodzi w zakres dokumentu, a co nie wchodzi, ew. wskazanie na dokumenty powiązane}
Udokumntowanie działa aplikacji z perspektywy użytkownika łącznie z udokumentowaniem działanie kodowania Hamminga.

### 1.3 Odbiorcy

{określenie adresatów dokumentu, może być to typ odbiorcy; tu: zleceniobiorca (Katedra), członkowie zespołu projektowego oraz wymienione z nazwiska osoby, do których dokument ma dotrzeć}
Zleceniodawca - dr inż. Bartosz Czaplewski (KSTI)
Członkowie zespołu:
Bartosz Kołakowski
Michał Mróz
Paweł Jastrzębski
Maksym Nowak
Piotr Noga

### 1.4 Terminologia

{wyjaśnienie używanych w dokumencie pojęć i skrótów, oznaczenia używane wewnątrz dokumentu np. oznaczenia wymagań}
GUI (graphical user interface) – interfejs graficzny
Frontend – kod odpowiedzialny za wyświetlanie/prezentację danych, zawarty w plikach. qml
Backed – kod kodujący wiadomości i poprawiający błędy przekazane przez frontend; implementacja kodowania. Zawarty w plikach .cpp
GF – ciało skończone/ciało Galois (Galois Field)

## 2 Dokumentacja techniczna projektu

{o zakresie dokumentacji decyduje opiekun, tu należy zacząć opis rozwiązania technicznego wg zaleceń opiekuna, w układzie redakcyjnym najlepiej oddającym charakter projektu – kolejne rozdziały, podrozdziały, punkty}

### 2.1 Ogólne:

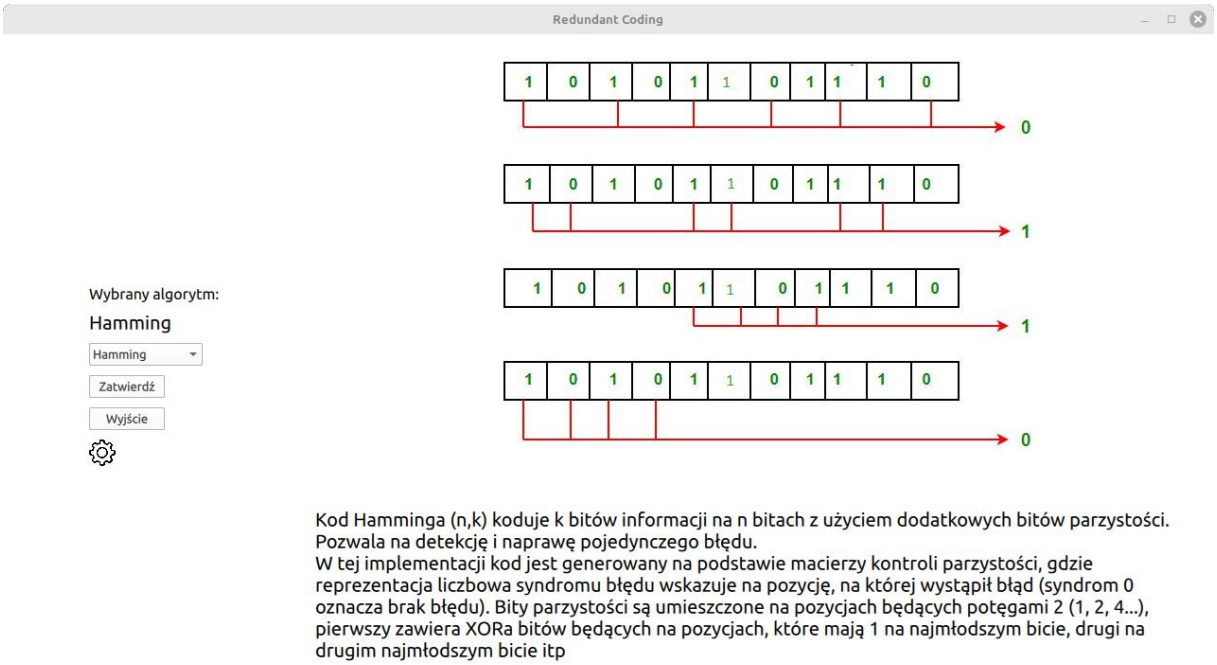
Projekt jest wykonany w języku C++, we frameworku Qt. Używane są 2 rodzaje plików – typowe pliki .cpp, w których jest zawarty „backend” aplikacji oraz pliki. qml; „frontend”, zawierają informacje o tym, co tak naprawdę ma się wyświetlić na ekranie użytkownika. Preferowanym środowiskiem developerskim jest program Qt Creator. Projekt można rozwijać zarówno na systemie Linux, jak i na Windows. W celu rozwijania aplikacji, trzeba zainstalować na systemie bibliotekę Boost. Wszelkie inne używane biblioteki (jak GoogleTest) są instalowane automatycznie.

Końcowa aplikacja jest eksportowana jako plik .exe. W celu generacji, najpierw trzeba zbudować projekt w QtCreatorze (najlepiej na profilu release, nie debug), co wygeneruje katalog o nazwie zbliżonej do „build-Redundant-Coding-Visualization-Desktop_Qt_6_6_2_MinGW_64_bit-Release”. Następnie należy włączyć konsolę MinGW (powinno dać się wyszukać, nazwa w stylu „Qt 6.6.2 (MinGW 11.2.0 64-bit)”), przejść w niej do tego katalogu i wykonać polecenie „windeployqt appRedundantCoding.exe”. Należy rozpowszechniać cały katalog (np. jako zip), a nie sam plik exe, gdyż w katalogu znajdują się zasoby potrzebne do uruchomienia – jak biblioteki dll.

## 2.2 Frontend/Wygląd:

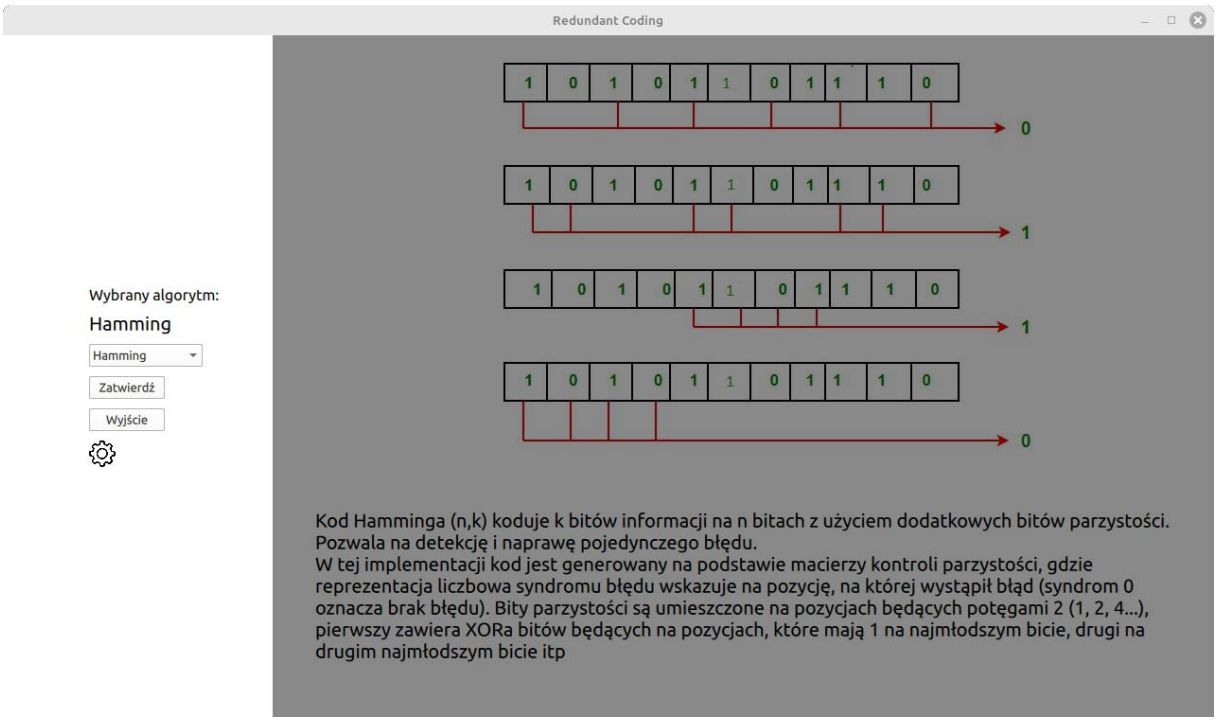
Przedstawiony zostanie wygląd poszczególnych sekcji programu, tj. menu głównego, wprowadzania danych dla danego algorytmu, przedstawienia jego działania. Wyjaśnione zostaną elementy, z których składa się każda ze stron występujących w programie.

### 2.2.1 Menu główne




*Rysunek 1 Wygląd menu głównego programu*

Po uruchomieniu programu, użytkownikowi zostaje przedstawione menu główne (Rysunek 1), którego kod źródłowy znajduje się w pliku „src_gui/Main.qml”. Składa się ono z dwóch obszarów – „Opisowego” i „Interaktywnego”.

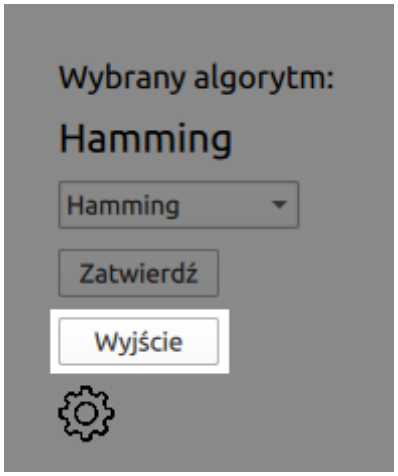



*Rysunek 2 Wyróżniony obszar „Interaktywny”*

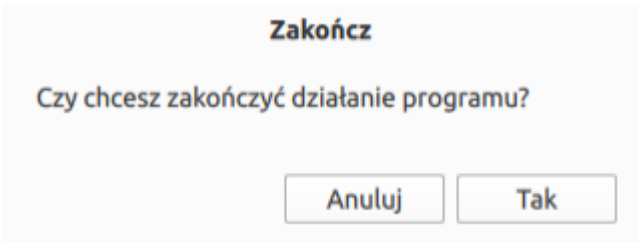
Obszar „Interaktywny” (Rysunek 2) zawiera trzy przyciski – „Ustawienia”, „Wyjście” oraz „Zatwierdź”, listę rozwijaną oraz tekst.


*Rysunek 3 Przycisk "Ustawienia"*

Przycisk „Ustawienia” przedstawiony za pomocą ikony zębatki (Rysunek 3), umożliwia przejście do okna z ustawieniami programu takimi jak wybór języka programu. Ów okno zostanie przedstawiona w dalszej części dokumentu.




*Rysunek 4 Przycisk "Wyjście"*




*Rysunek 5 Potwierdzenie zamknięcia programu*

Przycisk „Wyjście” (Rysunek 4) powoduje wywołanie okienka (Rysunek 5), w którym użytkownik zostaje zapytany o potwierdzenie swojej decyzji o zamknięciu programu.


*Rysunek 6 Przycisk "Zatwierdź"*

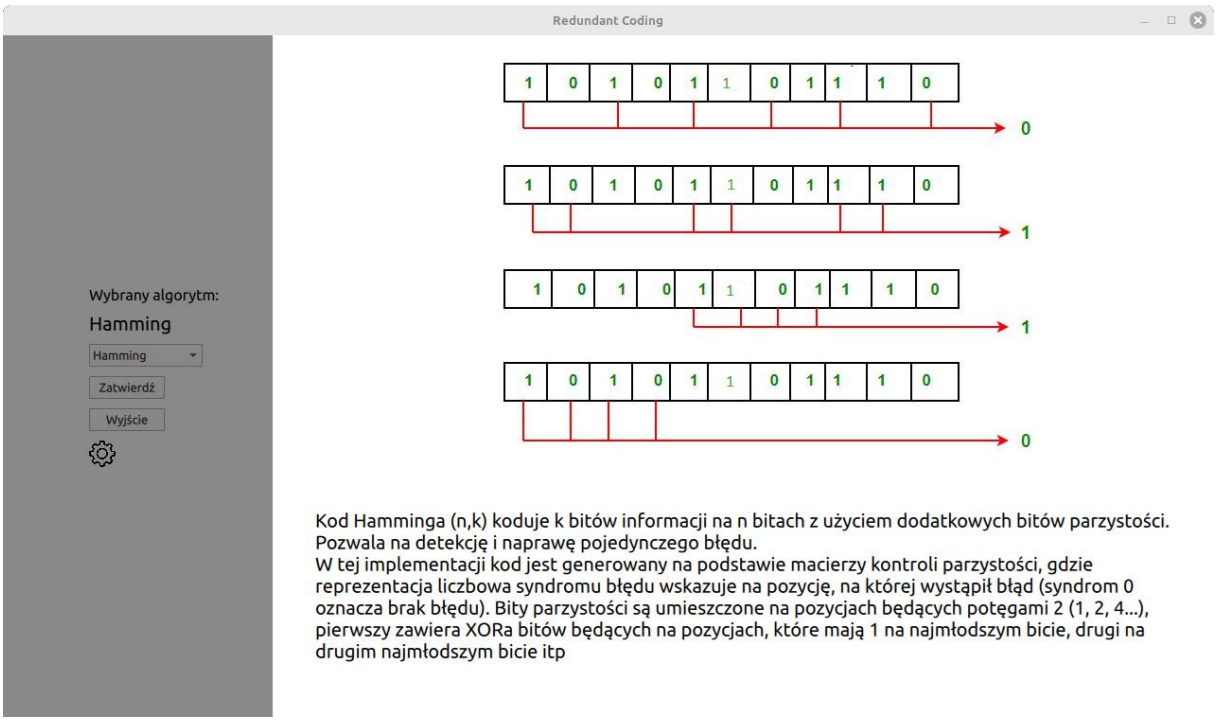

*Rysunek 7 Tekst informujący o wybranym algorytmie*


*Rysunek 8 Rozwijana lista*

Ostatnim przyciskiem widocznym w menu głównym jest „Zatwierdź” (Rysunek 6). Po kliknięciu w niego, program przechodzi do kolejnego okna programu, nazwanej dalej „Startowe dane”, w której następuje wprowadzenie danych, na których będzie operował program.

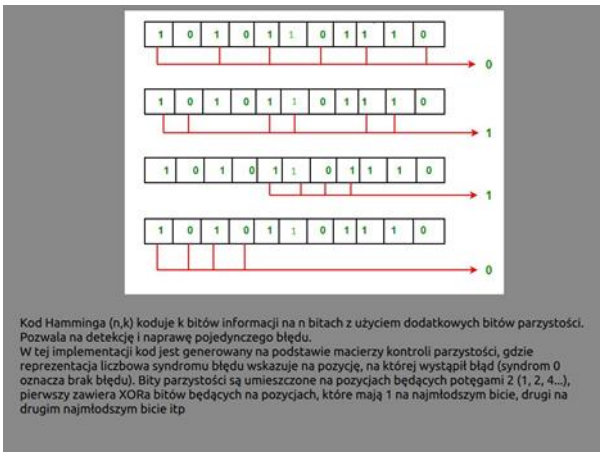
Nad przyciskami znajduje się rozwijana lista (Rysunek 7), zawierająca do wyboru algorytmy, które będą przedstawiane w późniejszym etapie działania programu. W obecnym stanie aplikacji, do wyboru jest wyłącznie jeden działający algorytm – algorytm Hamminga oraz dwie niedziałające opcje, które mają na celu jedynie wizualizację możliwości wyboru w przyszłości większej liczby algorytmów.

Ze wspomnianą listą, powiązany jest znajdujący się nad nią tekst (Rysunek 8), który wyświetla, jaki algorytm w danym momencie został wybrany.

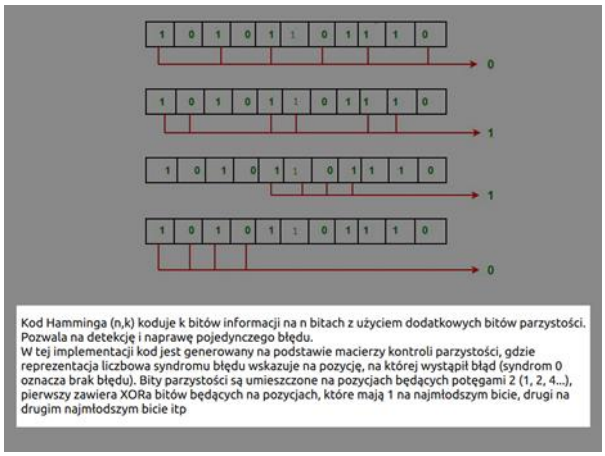



*Rysunek 9 Wyróżnienie obszaru "Opisowego"*

Obszar „Opisowy” (Rysunek 9) ma za zadanie przybliżyć użytkownikowi wybrany przez niego w obszarze „Interaktywnym” algorytm. Jest on zbudowany z dwóch elementów – obrazu oraz opisu.



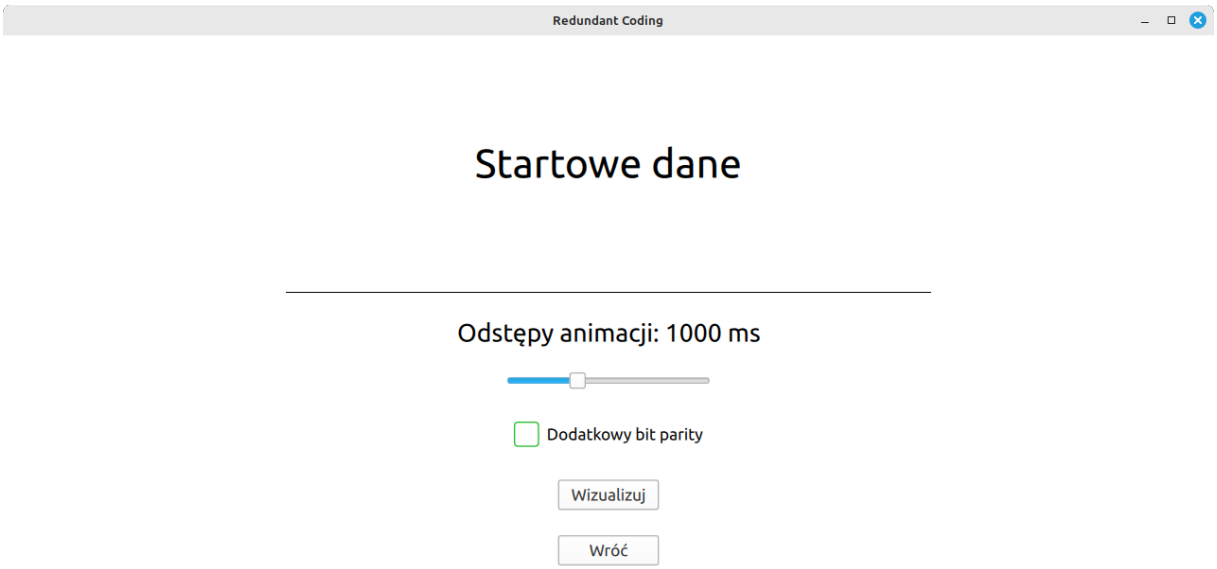
*Rysunek 10 Graficzna prezentacja algorytmu*



*Rysunek 11 Opis algorytmu*

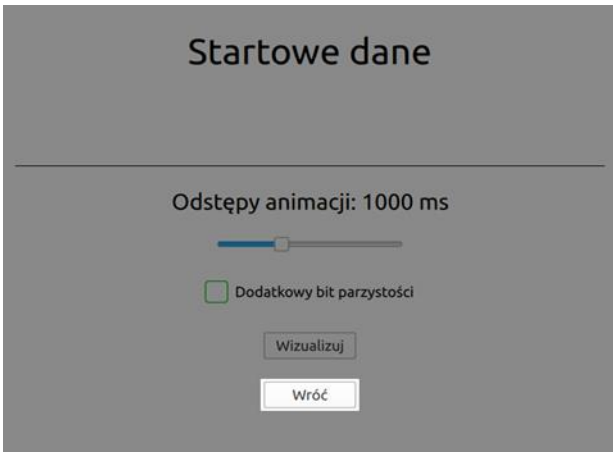
Obraz (Rysunek 10) przedstawia wizualizację wybranego algorytmu, natomiast opis (Rysunek 11) wyjaśnia jego działanie.

### 2.2.2 Startowe dane

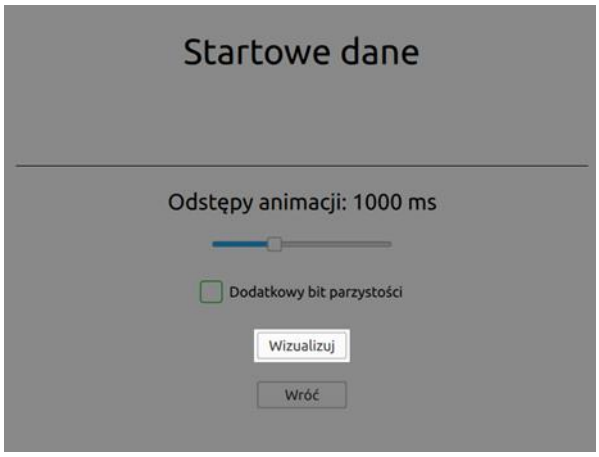



*Rysunek 12 Okno "Startowe dane"*

Kolejnym oknem w aplikacji jest „Startowe dane” (Rysunek 12), który umożliwia wprowadzenie danych, na których program będzie operował. Zawiera ono dwa przyciski - „Wróć” i „Wizualizuj”, przycisk typu Checkbox, suwak, tekst informujący o odstępach animacji, pole tekstowe oraz tytuł okna.



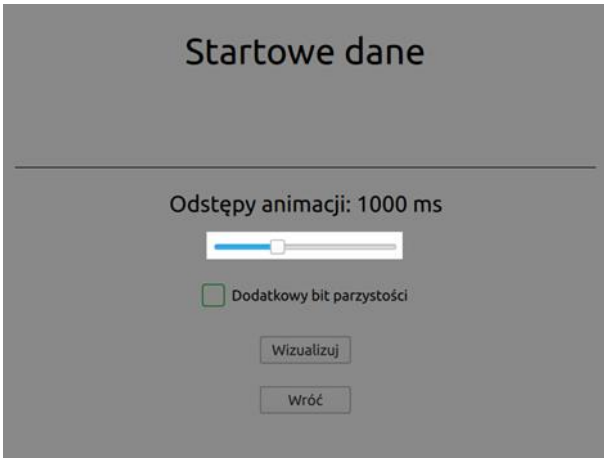

*Rysunek 13 Przycisk "Wróć"*




*Rysunek 14 Przycisk "Wizualizuj"*

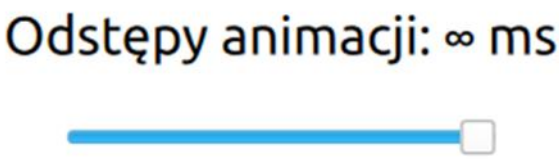
Przycisk „Wróć” (Rysunek 13) umożliwa powrót do menu głównego, natomiast przycisk „Wizualizuj” (Rysunek 14) przechodzi do głównej części aplikacji, czyli wizualizacji algorytmu.


*Rysunek 15 Przycisk typu checkbox*




*Rysunek 16 Suwak*

„Dodatkowy bit parzystości” (Rysunek 15) wprowadza w algorytmie dodatkowy bit parzystości. Suwak (Rysunek 16) ustawia czas jaki trzeba odczekać, by wykonał się kolejny krok algorytmu. Możliwy do ustawienia czas jest w przedziale (0 sekund - 2,9 sekund).




*Rysunek 17 Wartość ∞ na suwaku*

Jest też możliwość włączenia przechodzenia do kolejnego kroku algorytmu poprzez przycisk. W tym przypadku należy na suwaku ustawić specjalną wartość ∞ (Rysunek 17), która znajduje się na końcu skali suwaka z prawej strony.


*Rysunek 18 Pole tekstowe*


*Rysunek 19 Odstępy animacji*

Odstępy animacji (Rysunek 18) informują użytkownika o tym, jaki jest potrzebny czas do odczekania, by przejść do kolejnego kroku animacji prezentującej działanie algorytmu. Czas jest podany w milisekundach. Pole tekstowe (Rysunek 19) umożliwia wprowadzenie ciągu

liczbowego w formie binarnej, który następnie zostanie zakodowany nadmiarowo. Przyjmuje ono wyłącznie zera i jedynki. Pozostałe znaki są pomijane, zatem aplikacja jest zabezpieczona przed wprowadzeniem nieprawidłowego ciągu znakowego.

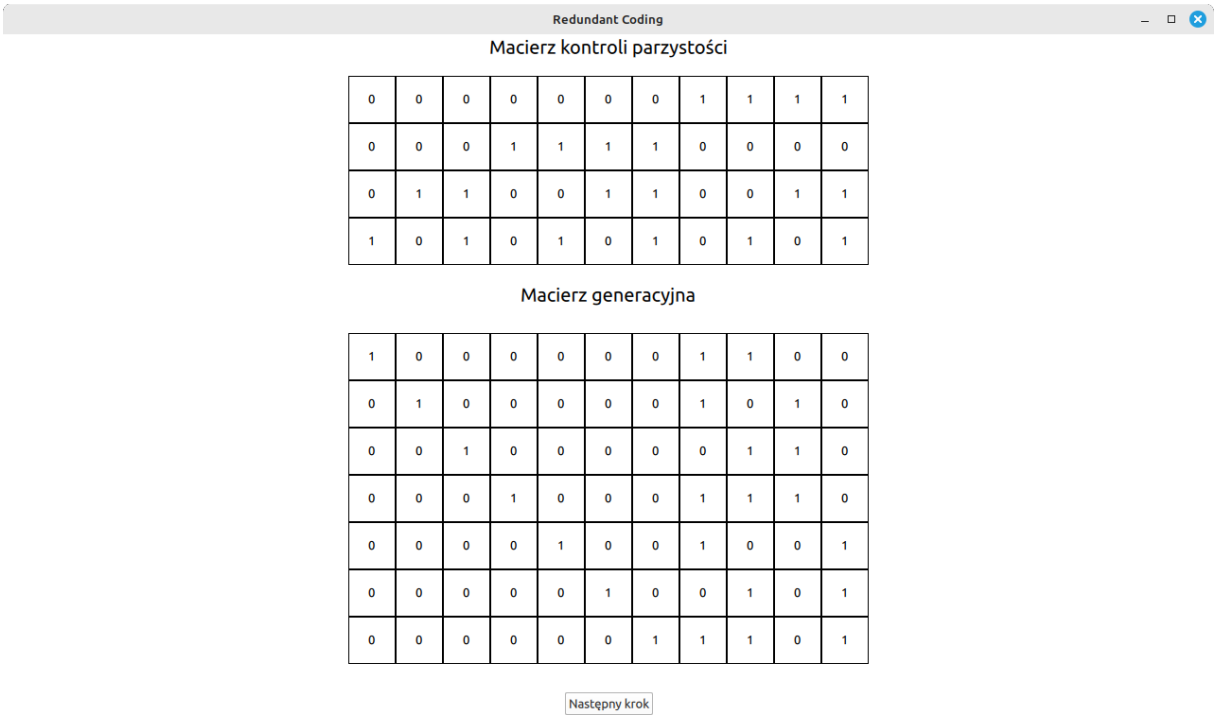

*Rysunek 20 Tytuł okna*

Ostatnim widocznym elementem okna jest jego tytuł (Rysunek 20).

### 2.2.3 Algorytm Hamminga

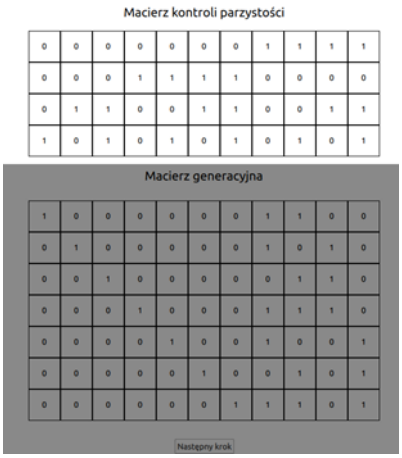
W kolejnych podrozdziałach będą przedstawione wszystkie okna występujące przy działania algorytmu, w tym poszczególne elementy tych okienek.

### 2.2.3.1 Macierze

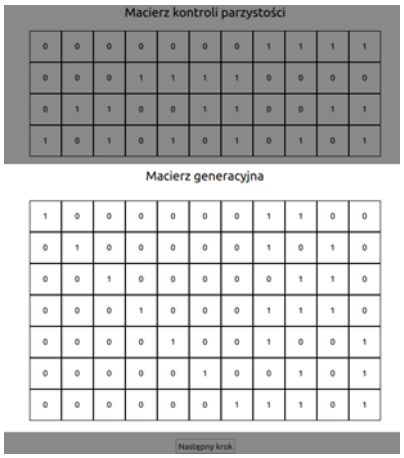



*Rysunek 21 Okno "Macierze"*

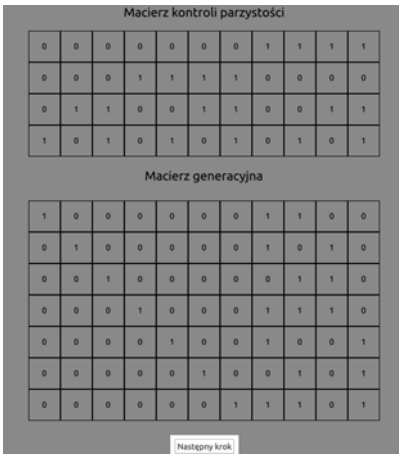
Kolejnym oknem w programie jest okno „Macierze” (Rysunek 21).




*Rysunek 22 Macierz kontroli parzystości*




*Rysunek 23 Macierz generacyjna*




*Rysunek 24 Przycisk "Następny krok"*

Przedstawia ono przede wszystkim macierz kontroli parzystości (Rysunek 22) oraz macierz generacyjną (Rysunek 23). Kod odpowiedzialny za to okno znajduje się w pliku „src-gui/HammingGenerationMatrix.qml”. Obie macierze przedstawione w tym oknie generowane są dynamicznie na podstawie ciągu liczbowego podanego przez użytkownika w poprzednim oknie. Tworzenie ich w formie graficznej zostało przedstawione w rozdziale Wizualizacja macierzy. Poza macierzami, w oknie znajdują się jeszcze tytuły tych macierzy, które znajdują się nad nimi. Na samym dole okna znajduje się przycisk „Następny krok” (Rysunek 24), który sprawia, że użytkownik przechodzi do okna demonstrującego działanie danego algorytmu.

#### 2.2.3.2 Plik Hamming.qml

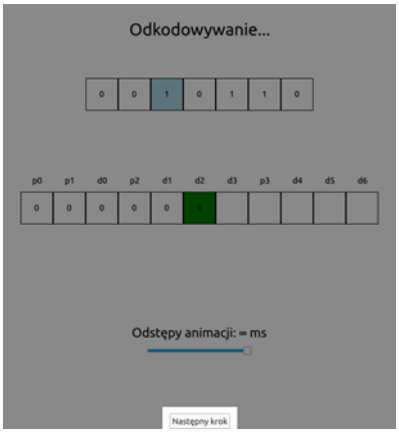
Po wpisaniu ciągu liczbowego w oknie „Startowe dane” i przejścia dalej, znaczna część programu związana z wizualizacją przebiegu algorytmu Hamminga znajduje się w pliku „src_gui/Hamming.qml”. Jest on najbardziej zaawansowanym oknem związanym z tym algorytmem, gdyż poza ostatnim oknem „”, cała dalsza część aplikacji jest wykonywana w oparciu o ten właśnie plik. W kolejnych rozdziałach, jeśli nie będzie wspomniane inaczej, dana sekcja będzie dotyczyła zawartości „src_gui/Hamming.qml”.

#### 2.2.3.3 Odkodowywanie

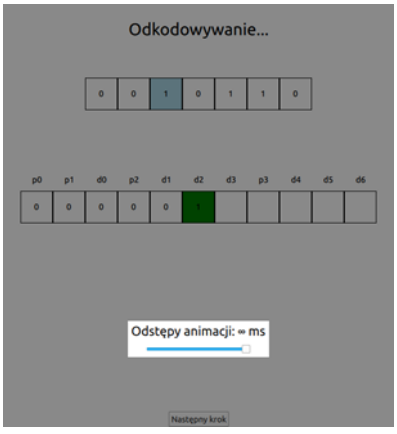

*Rysunek 25 Okno "Odkodowywanie" przed zapisaniem macierzy nadmiarowej*


*Rysunek 26 Okno "Odkodowywanie" po zapisaniu macierzy nadmiarowej*

Właściwa prezentacja działania algorytmu rozpoczyna się właśnie w tym oknie (Rysunek 25 i Rysunek 26). Okno to w tym momencie zbudowane jest z dwóch macierzy, suwaka oraz przycisku „Następny krok”, jeśli suwak został ustawiony na wartość ∞.



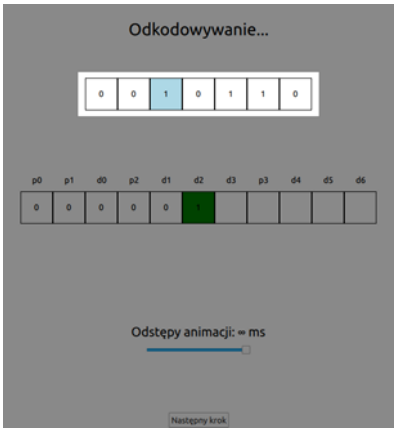

*Rysunek 27 Przycisk "Następny krok"*




*Rysunek 28 Suwak*

Jeśli w oknie jest widoczny przycisk „Następny krok” (Rysunek 27), umożliwia on przejście do kolejnego kroku w działaniu algorytmu. Suwak (Rysunek 28) tak jak w oknie „Startowe dane” ustawia czas między krokami algorytmu.


*Rysunek 29 Macierz kodowania nadmiarowego*




*Rysunek 30 Macierz oryginalnego ciągu liczbowego*

Macierzą znajdującą się na środku okna jest macierz kodowania nadmiarowego (Rysunek 29). Nad poszczególnymi komórkami macierzy znajdują się ich oznaczenia, „pX", bądź „dX". „p" oznacza bit, który jest bitem parzystości, natomiast „d" bitem z ciągu liczbowego. Jako „X" należy rozumieć numer bitu w danym jego rodzaju. Przykłady notacji wyglądają następująco:

- „p3" należy odczytać jako: bit parzystości będący trzeci w kolejności.
- „d5" oznacza odpowiednio: bit ciągu liczbowego, będący piąty w kolejności.

Warto nadmienić, że numer „X" **nie oznacza**, że bit znajduje się na danej pozycji w macierzy. Przykładowo bit „d5" widoczny na Ilustracji 29 nie znajduje się na pozycji numer 5, tylko numer 9, gdyż poprzedzają go również bity parzystości od 0 do 3.

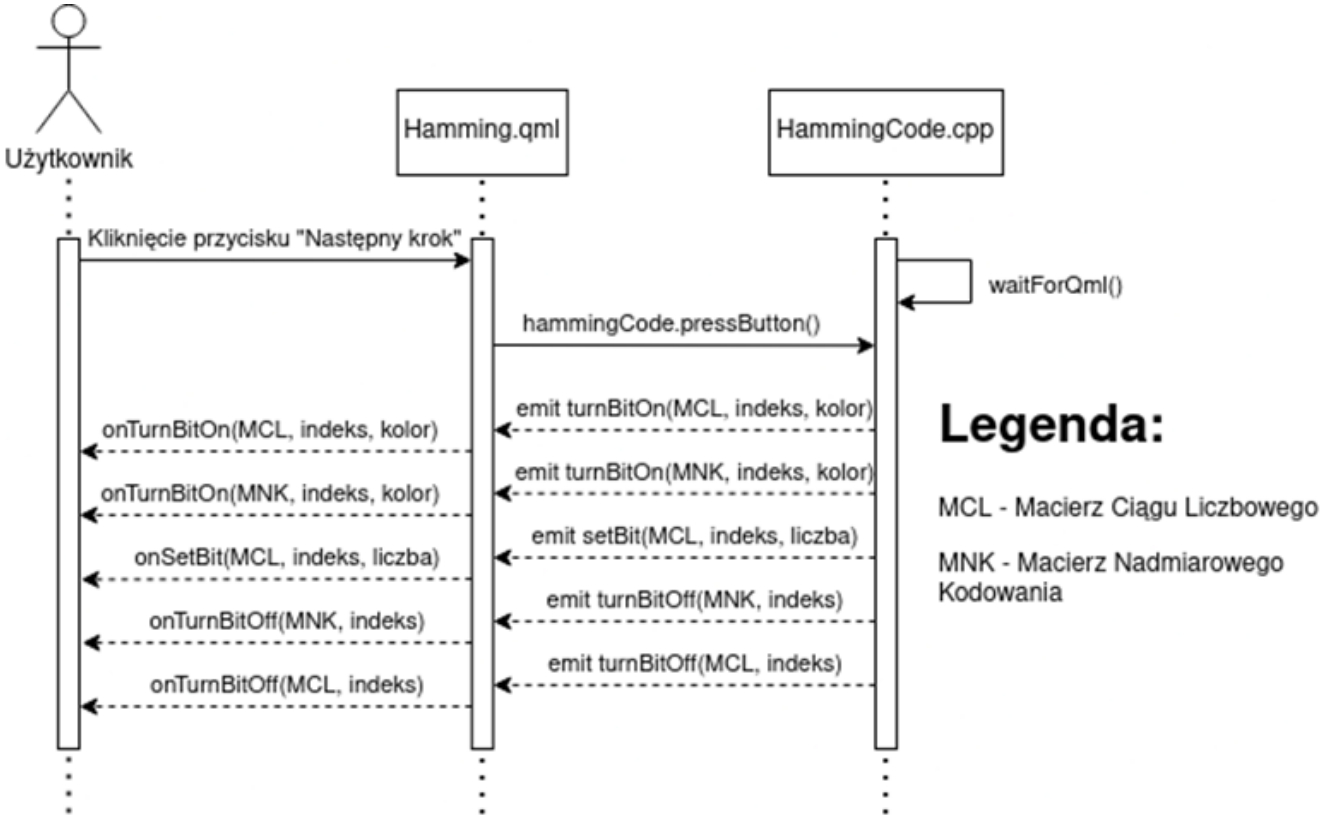
Nad macierzą kodowania nadmiarowego znajduje się macierz z ciągiem liczbowym, wprowadzonym wcześniej przez użytkownika (Rysunek 30).

Ogólny sposób wizualizacji macierzy został przedstawiony w rozdziale „Wizualizacja macierzy", natomiast w tym oknie stosowany jest również dodatkowy zabieg wobec macierzy nadmiarowego kodowania. Jak można zauważyć na Ilustracjach 25 i 26, macierz na początku nie jest wypełniona, by na końcu była już w pełni zapisana. Sposób wypełniania został przedstawiony w rozdziale „Wypełnianie macierzy w Hamming.qml".

#### 2.2.3.4 Wypełnianie macierzy w Hamming.qml

Za wypełnianie macierzy w tym etapie działania programu odpowiada backendowa funkcja *encodeDataAsync* znajdująca się w pliku „src/HammingCode.cpp". Wykorzystuje ona zawarte w pliku obecnego okna funkcje *onTurnBitOn, onTurnBitOff* oraz *onSetBit*. Za pomocą funkcji *onTurnBitOn* backendowa funkcja *encodeDataAsync* ustawia odpowiedni kolor dla danej komórki macierzy, tj. dla bitów będących bitami parzystości ustawia kolor czerwony, a dla bitów oznaczających kolejny bit z ciągu liczbowego, kolor zielony. Funkcja *onSetBit* umożliwia wpisanie wartości danego bitu do macierzy. *OnTurnBitOff* ustawia z powrotem na kolor biały tło danej komórki macierzy, gdy należy przejść do kolejnej komórki.

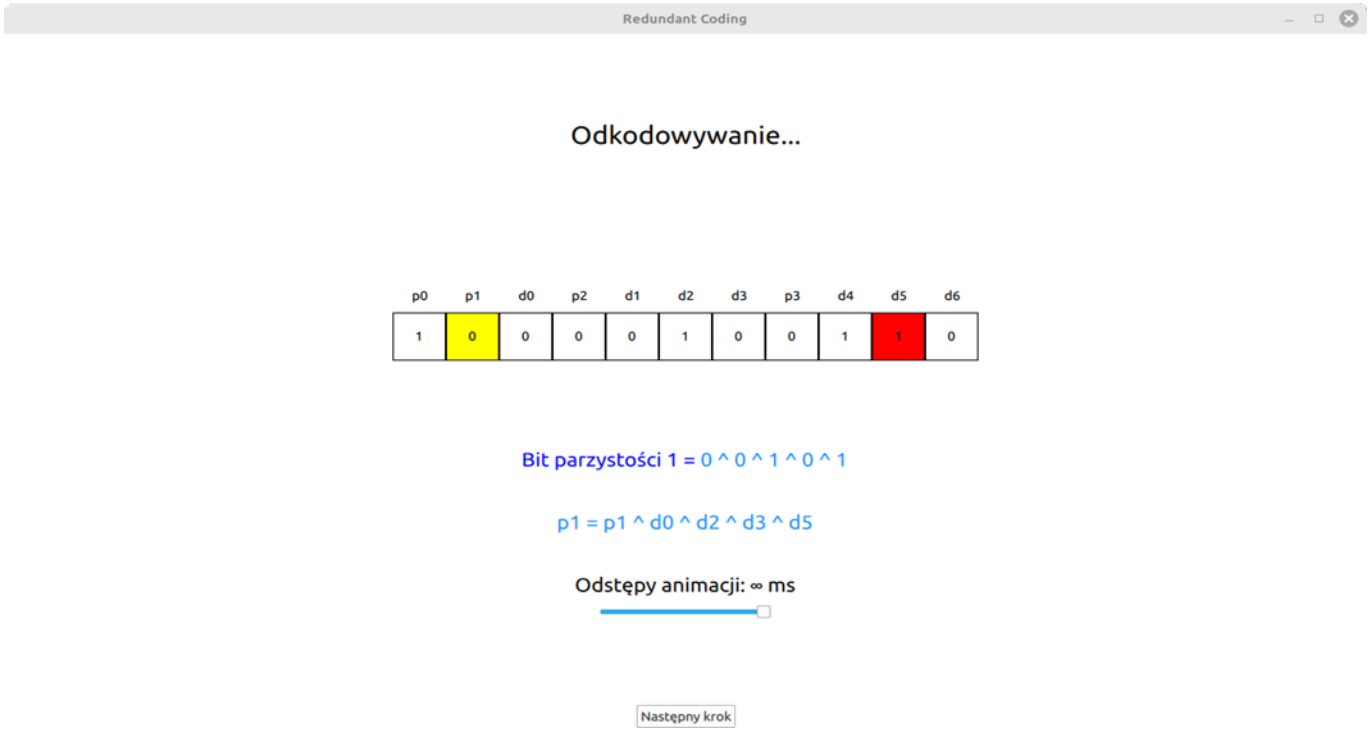
W trakcie wpisywania kolejnych bitów do macierzy kodowania nadmiarowego, wspomniana wyżej backendowa funkcja *encodeDataAsync* zmienia również kolor tła odpowiednich komórek macierzy ciągu liczbowego, na dokładnie tych samych zasadach, co wcześniej, z pominięciem wpisywania wartości bitów za pomocą *onSetBit*, tj. *onTurnBitOn* zmienia kolor tła na jasnoniebieski a *onTurnBitOff* cofa tę zmianę.




*Rysunek 31 Uproszczony diagram sekwencji animacji wypełniania komórki macierzy nadmiarowego kodowania wraz z animacją macierzy ciągu liczbowego*

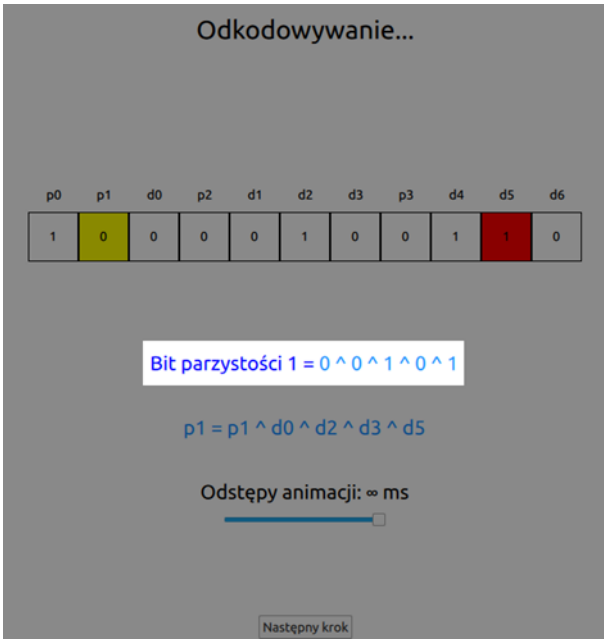
Na Rysunek 31 zamieszczony został uproszczony diagram sekwencji, w którym jest zaprezentowany sposób, w jaki działa aplikacja przy animowaniu wypełniania danej komórki macierzy nadmiarowego kodowania.

#### 2.2.3.5 Obliczanie bitów parzystości

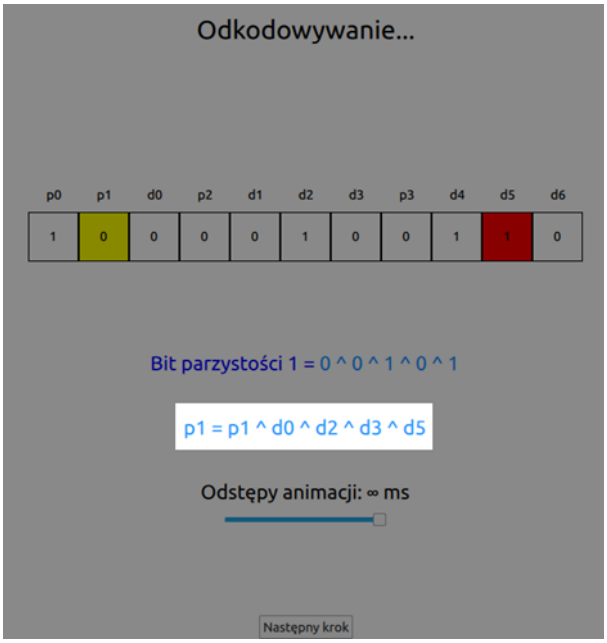



*Rysunek 32 Okno "Obliczanie bitów parzystości"*

Po wypełnieniu macierzy z nadmiarowym kodowaniem należy teraz obliczyć wartości poszczególnych bitów parzystości, gdyż przy wypełnianiu domyślnie były wpisywane zera. Samo okno się nie zmieniło, gdyż wciąż jest to „Odkodowywanie”, tak jak to głosi widoczny w nim tytuł, natomiast wygląd tego okna delikatnie różni się od poprzedniego (Rysunek 32). Z widoku zniknęła macierz ciągu liczbowego i została jedynie macierz z nadmiarowym kodem.

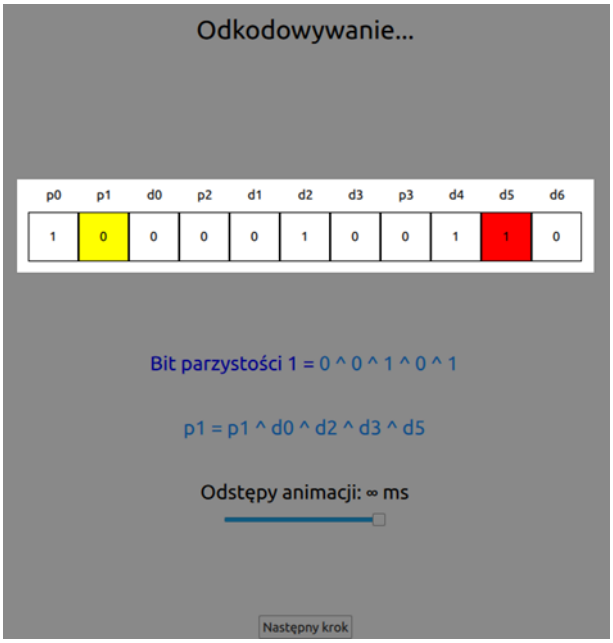



*Rysunek 33 Tekst "Obliczenia"*




*Rysunek 34 Tekst "Wzór"*

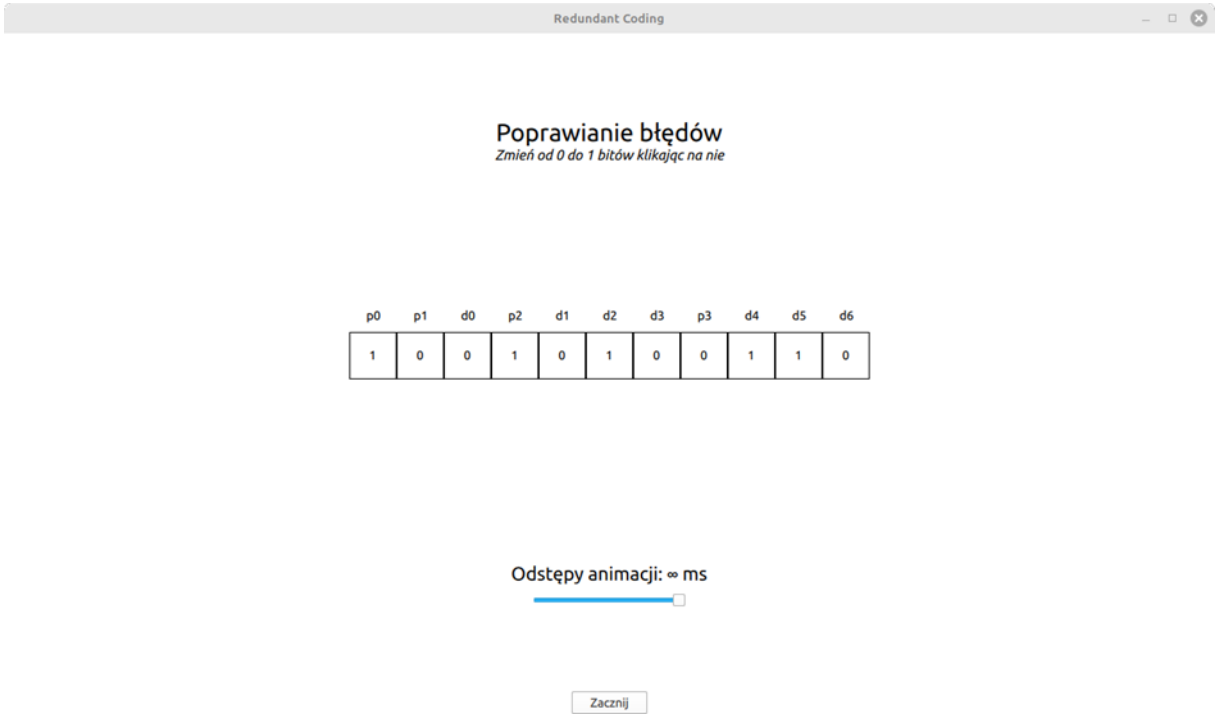
Pod nią pojawiły się dwa teksty. Pierwszy tekst „Obliczenia” (Rysunek 33) zawiera obliczenia rozpatrywanego bitu parzystości. Drugi tekst, nazwany „Wzór” (Rysunek 34) zawiera wzór, który został użyty w tekście „Obliczenia”.




*Rysunek 35 Animacja obliczania wartości bitu parzystości*

Oba teksty współgrają z animacją, która prezentuje obliczanie wartości bitu parzystości (Rysunek 35). Zmiana kolorów poszczególnych komórek działa na podobnej zasadzie, co zmienianie kolorów w macierzy ciągu liczbowego, które zostało opisane w rozdziale „Wypełnianie macierzy w Hamming.qml” z drobnymi różnicami, które zostaną teraz pokrótce przedstawione. W tym momencie to dwie komórki naraz mają zmieniony kolor. Rozpatrywany bit parzystości ma zmieniony kolor tła na żółty, natomiast bit brany w danym momencie pod uwagę, w zależności od swojej wartości, zmienia swój kolor na czerwony (gdy wartość wynosi 0) lub na zielony (gdy wartość wynosi 1).

#### 2.2.3.6 Poprawianie błędów



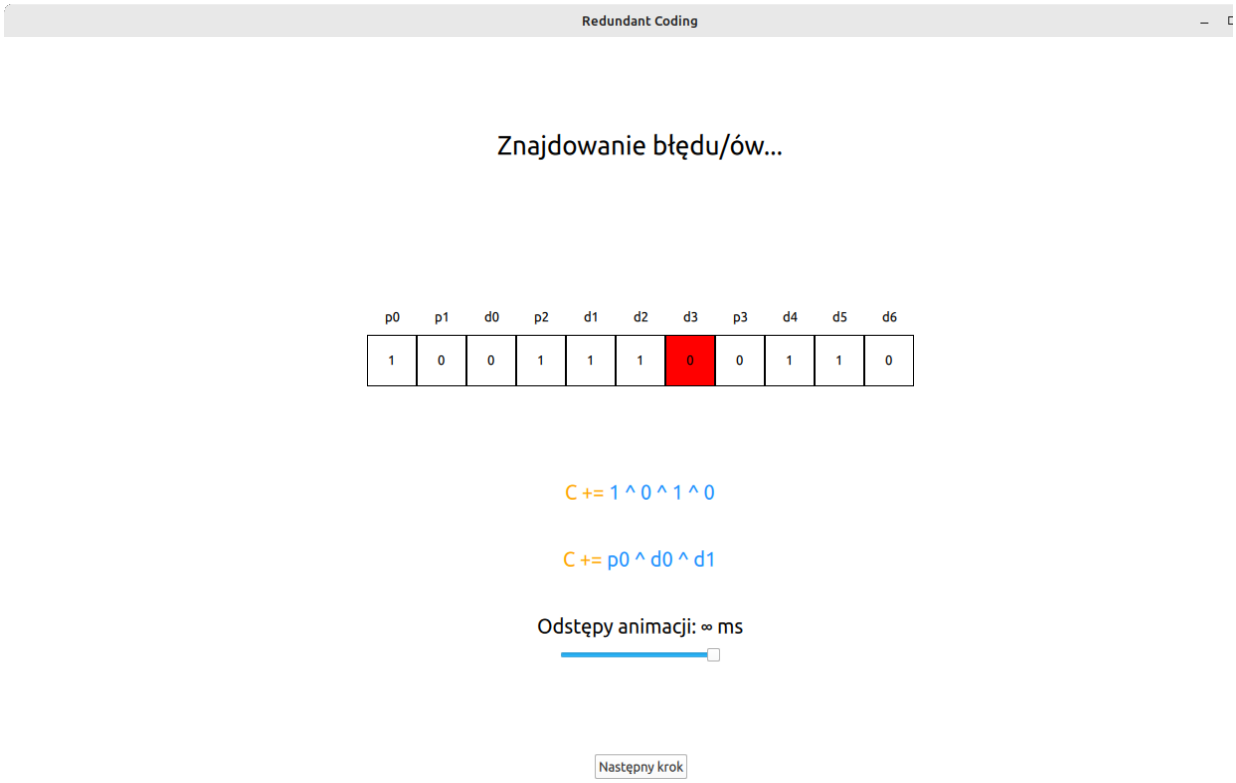

*Rysunek 36 Okno "Poprawianie błędów"*

Po zmianie bitów parzystości w macierzy nadmiarowego kodowania w poprzednim wyglądzie okna, użytkownik ma możliwość zmiany jednego bita (lub dwóch bitów, jeśli został zaznaczony dodatkowy bit parzystości). W rozdziale „Wizualizacja macierzy” zostało wspomniane, że struktura pojedynczego elementu, z którego składa się macierz, zawiera informację o obsłudze myszki. W tym momencie jest to wykorzystywane, dzięki czemu po kliknięciu we wskazaną komórkę macierzy, użytkownik zmienia jej wartość.


*Rysunek 37 Zmieniony bit o indeksie 5*

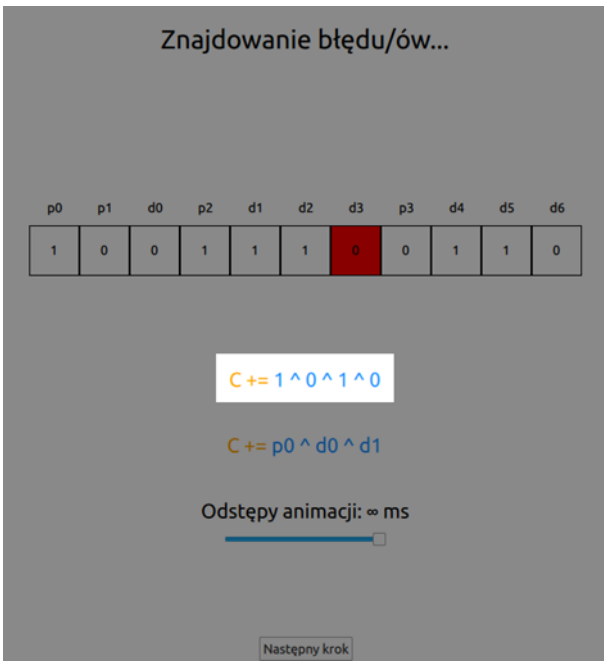
W celach demonstracyjnych został zmieniony bit „d1”, który znajduje się na pozycji o indeksie 5 (Rysunek 37).

### 2.2.3.7 Znajdowanie błędu/ów

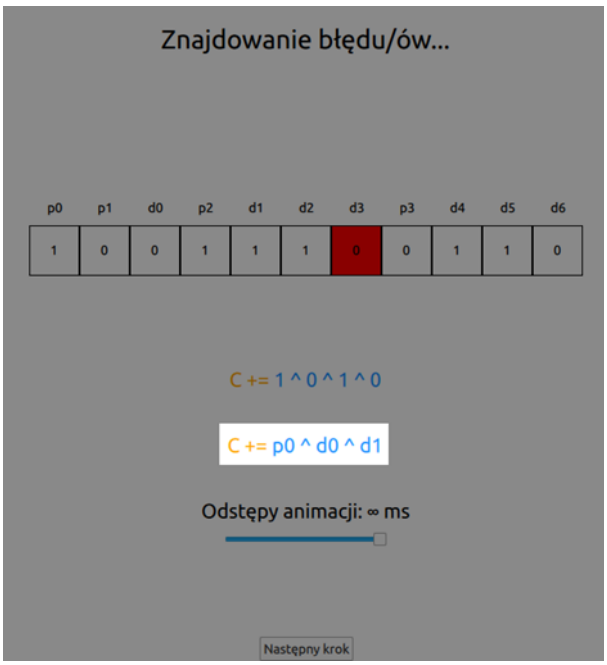



*Rysunek 38 Okno "Znajdowanie błędu/ów"*

Ten wygląd okna jest niemal taki sam jak w rozdziale „Obliczanie bitów parzystości” z drobną różnicą w tekstach znajdujących się pod macierzą i w animacji, gdzie zmieniany jest kolor tylko w jednej komórce macierzy w danym czasie.




*Rysunek 39 Tekst "Obliczenia"*



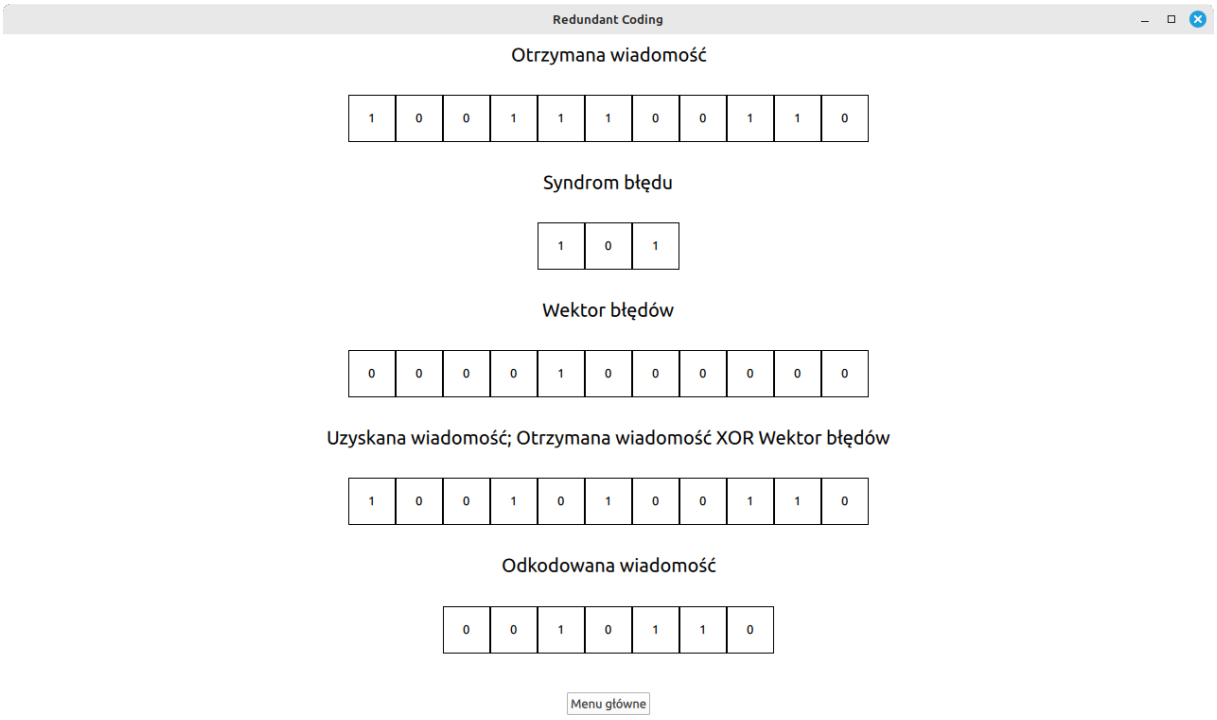

*Rysunek 40 Tekst "Wzór"*

Tak jak we wspomnianym rozdziale pierwszy tekst „Obliczenia” (Rysunek 39) zawiera obliczenia, ale tym razem obliczana jest liczba C. Liczba ta wskazuje na indeks, w którym został zmieniony bit przez użytkownika w poprzednim wyglądzie okna. Drugi tekst „Wzór” (Rysunek 40) zawiera zapis wzoru, który został wykorzystany w tekście „Obliczenia”.


*Rysunek 41 Ekran końcowy pliku "Hamming.qml"*

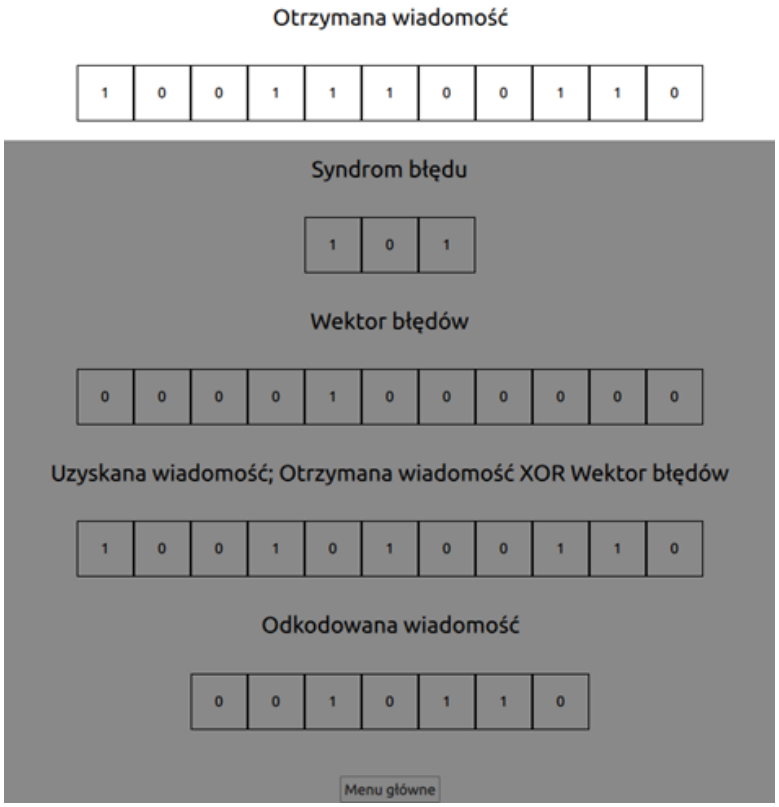
Po zakończeniu animacji wyszukiwania błędu, bądź błędów (Rysunek 41), w miejscu tekstu „Wzór”, pojawia się tekst informujący, na której pozycji jest błąd.

#### 2.2.3.8 Ekran końcowy




*Rysunek 42 Ekran końcowy algorytmu Hamminga*

Ostatnie okno, które pojawia się w trakcie prezentowania algorytmu Hamminga, znajduje się w pliku „src_gui/HammingSyndrome.qml”. Przedstawia ono 5 macierzy, tytuły nad nimi, co poszczególne macierze oznaczają oraz przycisk powrotu do menu głównego.

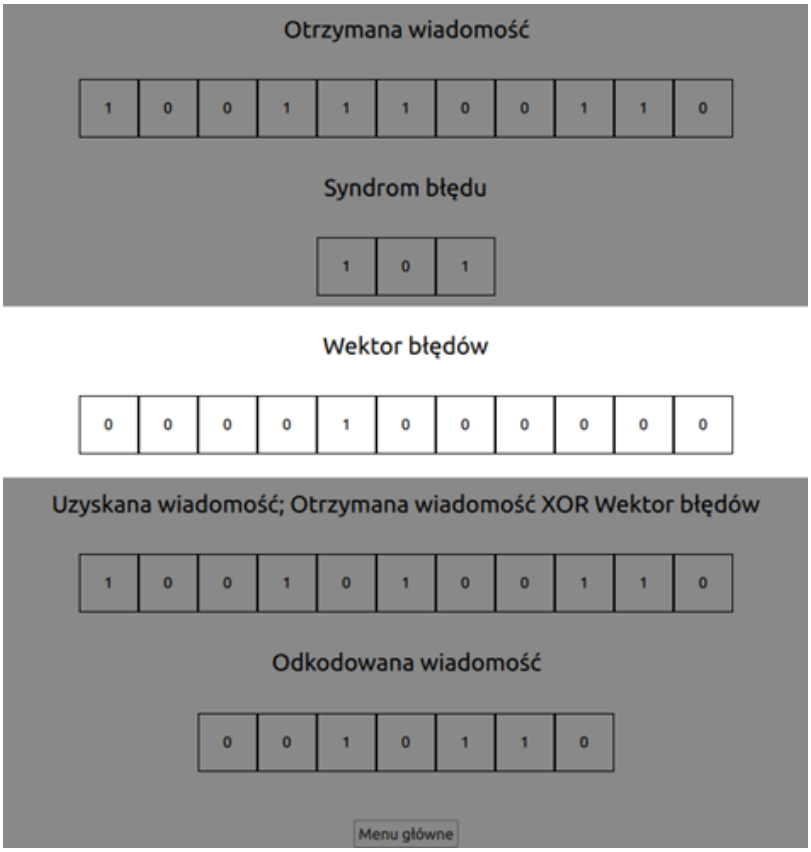



*Rysunek 43 Macierz "Otrzymana wiadomość"*

Macierz „Otrzymana wiadomość” (Rysunek 43) oznacza nadmiarowe kodowanie, w którym celowo został zmieniony bit przez użytkownika.

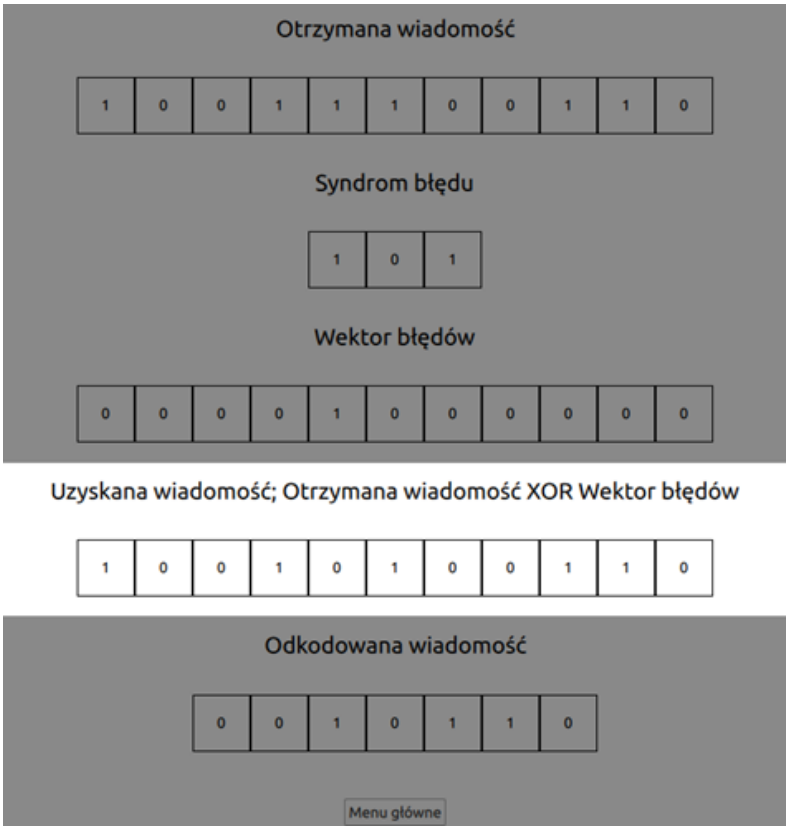

*Rysunek 44 Macierz "Syndrom błędu"*

„Syndrom błędu” (Rysunek 44) to macierz, w której zapisano w formie binarnej indeks pozycji, w której zawarty jest zmieniony bit przez użytkownika.



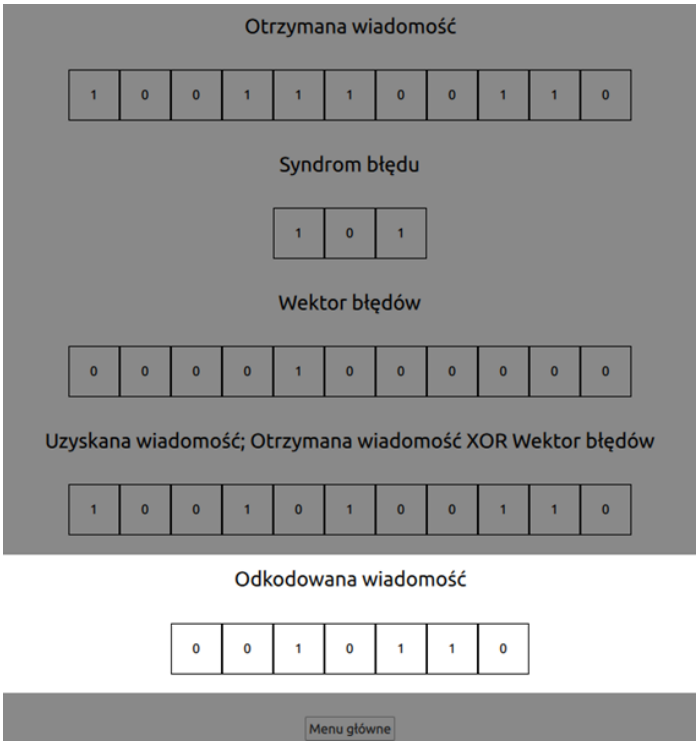

*Rysunek 45 Macierz "Wektor błędów"*

„Wektor błędów” (Rysunek 45) to macierz o tym samym rozmiarze, co oryginalna macierz nadmiarowego kodowania, która w całości jest wypełniona zerami, poza bitem o indeksie, o którym w oryginalnym kodzie nadmiarowym nastąpił błąd.



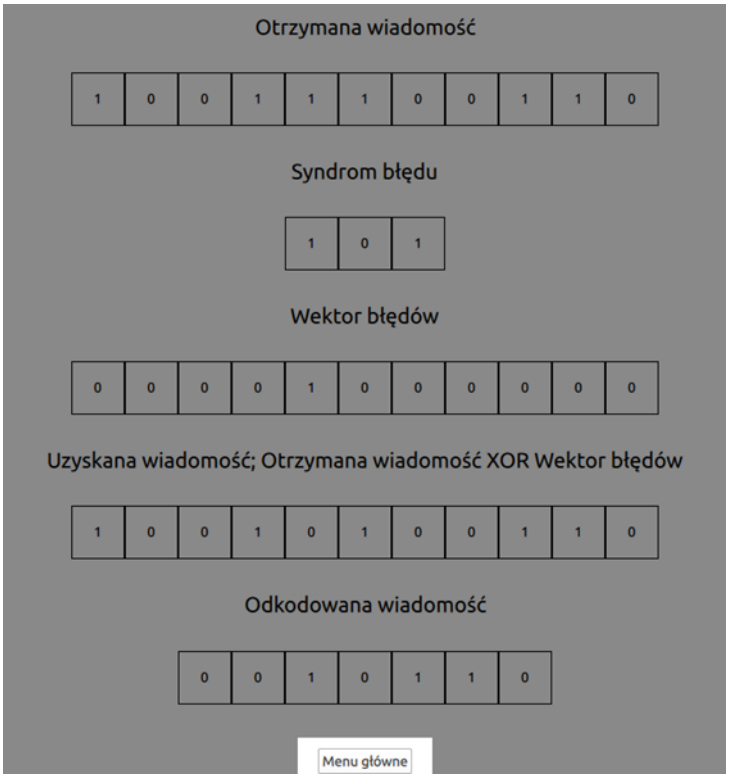

*Rysunek 46 Macierz "Uzyskana wiadomość"*

„Uzyskana wiadomość” (Rysunek 46) to macierz będąca wynikiem operacji alternatywy wyłączającej macierzy „Otrzymana wiadomość” z macierzą „Wektor błędów”.




*Rysunek 47 Macierz "Odkodowana wiadomość"*

Ostatnią z macierzy w tym oknie jest „Odkodowana wiadomość” (Rysunek 47), Jest to wynik działania algorytmu. Ciąg liczbowy powstały z bitów tej macierzy, powinien być równy temu, który został podany przez użytkownika na początku działania algorytmu, jeśli kodowanie nadmiarowe, wykrycie i naprawa błędu przebiegły prawidłowo.




*Rysunek 48 Przycisk "Menu główne"*

Na dole okna znajduje się przycisk „Menu główne” (Rysunek 48), który umożliwia przejście z powrotem do ekranu startowego.

#### 2.2.3.9 Wizualizacja macierzy

Macierz stanowi podstawę prezentowanego programu. W wielu oknach aplikacji generowane są one w różnych postaciach, czy to pojedynczego ciągu liczbowego, czy też w formie kompleksowych macierzy generacyjnych, zajmujących znaczną część okna, w którym się znajdują. Pliki odpowiedzialne za tworzenie ich w formie graficznej, widocznej dla użytkownika, znajdują się w folderze „src_gui/VisualizeComponents”. Proces tworzenia ich wizualizacji w pliku „ArrayRowLayout.qml” wygląda następująco:

1. Wywołując plik, zmieniana jest wartość zmiennej typu string *myArr*, która przechowuje ciąg liczbowy dla danego rzędu macierzy.
2. Na podstawie długości ciągu binarnego (ile zostało zawartych liczb w danym ciągu, w tym przypadku cyfr 0 i 1), będącego wpisanym we wspomnianej wyżej zmiennej, określana jest szerokość macierzy.
3. Funkcja Repeater powtarza rysowanie danych elementów tyle razy, ile wynosi wartość zmiennej *model*. Właśnie ta funkcja umożliwia graficzne przedstawienie danego rzędu macierzy. Zmienna *model* przyjmuje więc wartość szerokości macierzy.
4. Elementem, który jest cyklicznie rysowanym przez powyższą funkcję, jest typ *Rectangle*. Zawiera ona poszczególne informacje:

1. jakiego rozmiaru jest prostokąt, który reprezentuje daną komórkę macierzy,
2. tekst jaki jest zawarty w prostokącie, czyli liczba znajdująca się w danym miejscu,
3. w jaki sposób reaguje obszar wyrysowanego prostokąta po kliknięciu w niego myszką, co umożliwia zmienianie bitu zawartego w danej komórce macierzy (typ MouseArea oraz funkcja onClicked()).

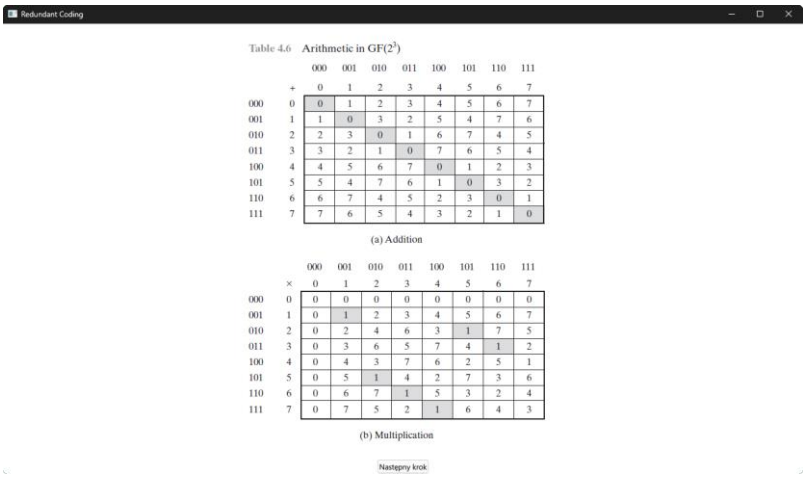
Ów element stanowi zatem komórkę danej macierzy, przy czym złożenie ich w funkcji Repeater stanowi rząd macierzy.

Na tym etapie rząd macierzy jest już widoczny na ekranie. Jeśli aplikacja ma za zadanie wyświetlić macierz składającą się z większej liczby rzędów, wywołuje ten plik tyle razy, ile wynosi jej wysokość, tak więc przykładowo dla macierzy mającej wysokość równą pięć, plik „ArrayRowLayout.qml" zostanie wywołany pięciokrotnie, sprawiając wrażenie, że wyrysowana w oknie macierz wygląda jednolicie.

### 2.2.4 Algorytm Reeda-Solomona

W kolejnych podrozdziałach przedstawione zostaną poszczególne ekrany występujące w trakcie działania algorytmu Reeda-Solomona. Wszystkie ekrany poza pierwszym znajdują się w pliku "ReedSolomon.qml", co będzie jeszcze zaznaczone w kolejnych podrozdziałach.

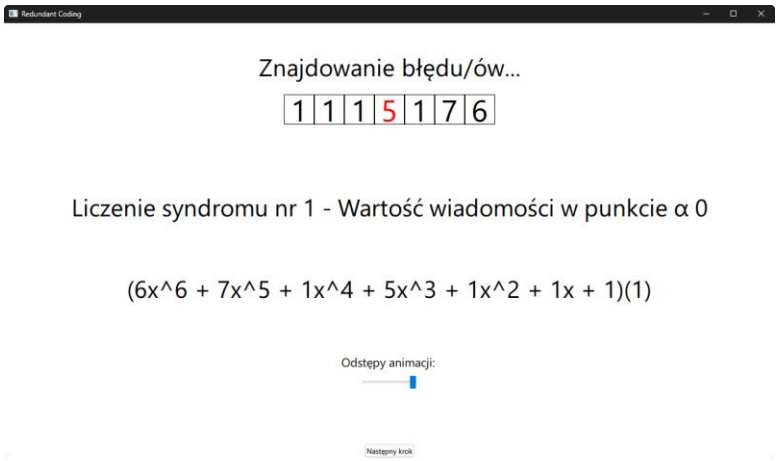
#### 2.2.4.1 Galois




*Rysunek Ekran Galois*

Jest to pierwszy ekran dostępny jedynie w algorytmie Reeda-Solomona. Przedstawia on arytmetykę stosowaną w dalszej części programu. Ekran składa się z dwóch części - dwóch tabel i przycisku "Następny krok". Jest to jedyny ekran, który **nie** znajduje się w pliku "ReedSolomon.qml". Jest on w "Galois.qml".

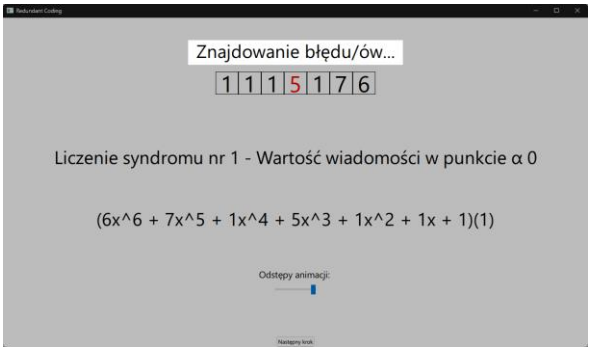
#### 2.2.4.2 Ogólne omówienie “ReedSolomon.qml”



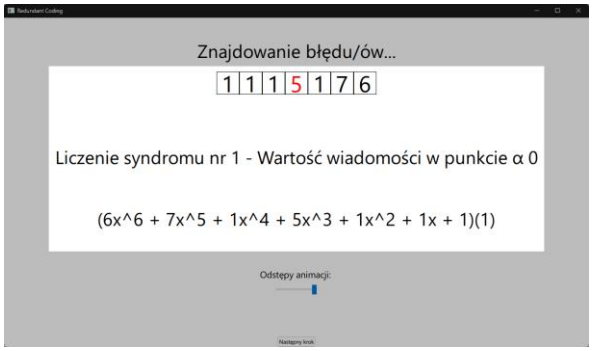

Rysunek Przykładowy ekran z pliku “ReedSolomon.qml”

Ekran składa się z następujących części: Tytuł (“Znajdowanie błędu/ów”), “obszar roboczy”, suwak (“Odstępy animacji:”) oraz w przypadku wybrania odstępu animacji na wartość ∞ przycisk “Następny krok”.

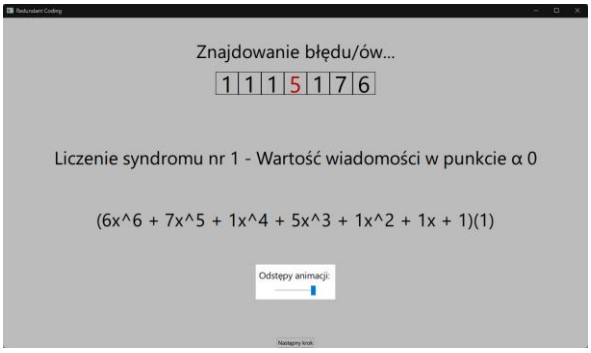
- Tytuł: Identyfikacja tego, jaka ogólna czynność odbywa się na ekranie, np. “Odkodowywanie”, “Znajdowanie błędu/ów”, “Szukanie pozycji błędu” itp.. (Rysunek)
- “Obszar roboczy”: pole, w którym przedstawiane jest szczegółowe działanie algorytmu, tj. pokazywane są wykorzystywane wzory, co w danej chwili jest obliczane, działania na liczbach. (Rysunek)
- Suwak “Odstępy animacji”: umożliwia ustawienie czasu pomiędzy kolejnymi krokami wykonywanymi w algorytmie. (Rysunek)
- Przycisk “Następny krok”: umożliwia przejście do następnej sekcji programu w przypadku, gdy została wybrana wartość ∞ odstępu animacji. (Rysunek)



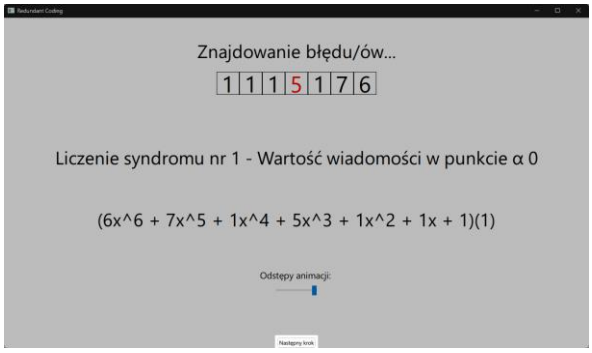

Rysunek Tytuł




Rysunek “Obszar roboczy”

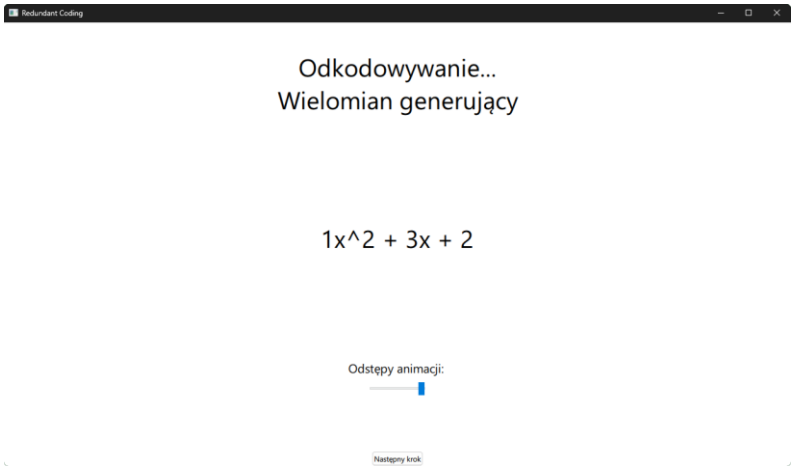



Rysunek Suwak “Odstępy animacji”




Rysunek Przycisk “Następny krok”

#### 2.2.4.3 Odkodowywanie

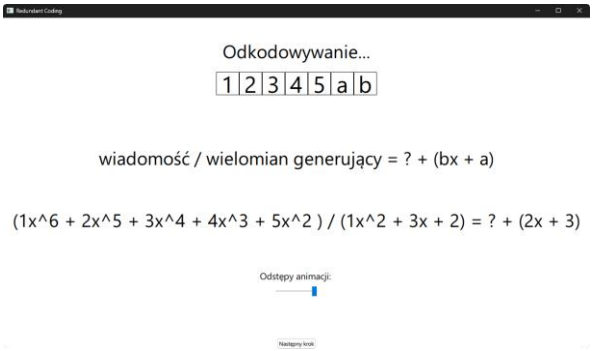



*Rysunek Ekran Odkodowywanie z podtytułem “Wielomian generujący”*

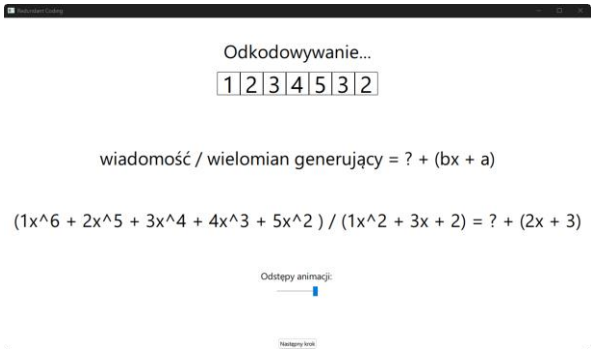
Na początku znajduje się wielomian generujący, który jest wykorzystywany w algorytmie. Wzór wielomianu został przedstawiony w “Obszarze Roboczym”.


Rysunek Przepisanie danych wejściowych

Następnie program przepisuje wartości podane przez użytkownika w ekranie “Startowe dane” do macierzy, na której będzie przeprowadzony algorytm Reeda-Solomona. Macierz ta pojawia się w górnej części ekranu, tuż pod tytułem.




Rysunek wiadomość / wielomian generujący przed wyliczeniem wartości liczb *a* i *b*




Rysunek Wyliczone wartości *a* i *b*

Pojawia się teraz ekran, w którym wyliczane są wartości *a* i *b*. Zrzut ekranu (Rysunek) po lewej stronie przedstawia sytuację sprzed wyliczenia tych wartości, natomiast na zrzucie ekranu po prawej stronie (Rysunek) widać już obliczone wartości.

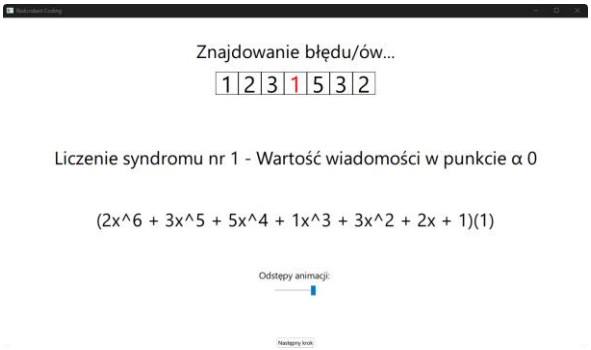
#### 2.2.4.4 Poprawianie błędów


Rysunek Poprawianie błędów z ustawionym błędem

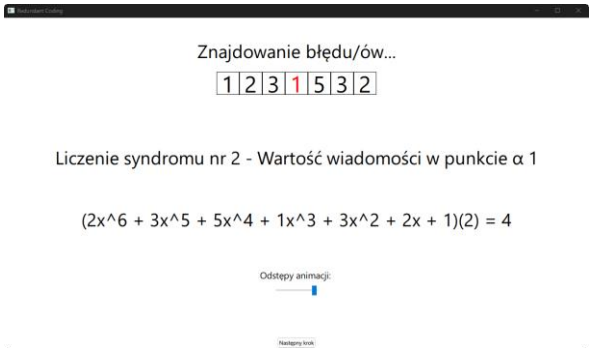
W tym ekranie pojawia się macierz wraz z dodatkowymi liczbami, które zostały wyliczone w poprzednim ekranie. Można wybrać, w której pozycji wiadomości ma się pojawić błąd, bądź też można zostawić ją bez zmian.

W dalszym omawianiu ekranów pokazany będzie przykład (wiadomość 12345), w którym użytkownik ustawił na czwartej pozycji błędną liczbę (zmienił prawidłową liczbę 4 na błędną liczbę 1). Zmieniona liczba jest wyróżniona czerwonym kolorem względem pozostałych liczb, pokolorowanych na czarno.

#### 2.2.4.5 Znajdowanie błędów

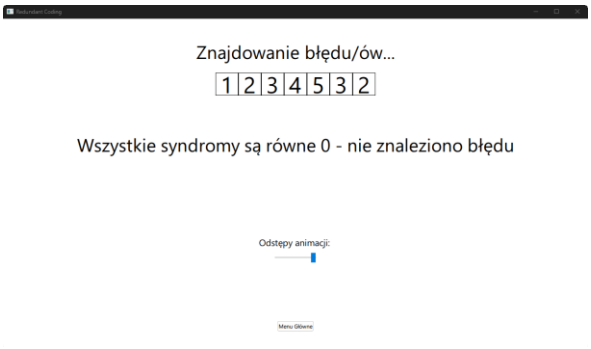



Rysunek Liczenie syndromu nr 1

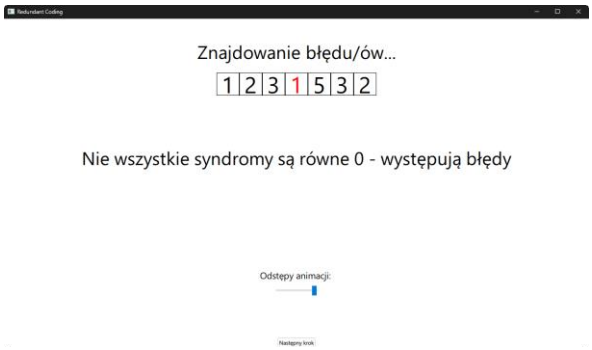



Rysunek Liczenie syndromu nr 2

W tym momencie program sprawdza, czy w wiadomości znajduje się błąd. Pokazane są liczenie poszczególnych syndromów.




Rysunek Prawidłowa wiadomość - Nie znaleziono błędów

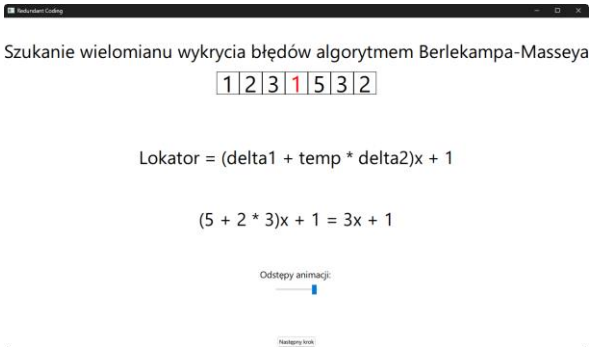



Rysunek Błędna wiadomość - Znaleziono błędy

W zależności od tego, czy program znalazł błąd w wiadomości, pojawia się odpowiednia informacja, co widać na powyższych zrzutach ekranu. Gdy użytkownik nie ustawił w wiadomości żadnego błędu, to algorytm kończy w tej chwili swoje działanie i po naciśnięciu przycisku “Menu główne” następuje powrót do ekranu startowego programu.

#### 2.2.4.6 Szukanie wielomianu wykrycia błędów


Rysunek Obliczanie delta 1

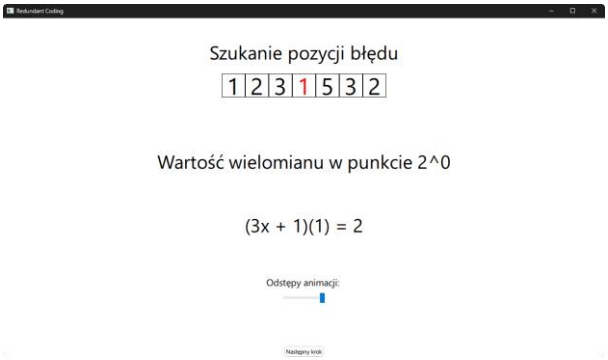



Rysunek Obliczanie Lokatora

W kolejnym ekranie następuje szukanie wielomianu wykrycia błędów. W “Obszarze roboczym” wypisywane są po kolei wykorzystywane wzory bądź zmienne z tych wzorów. Poza zmianami we wspomnianym “Obszarze roboczym” nic się nie zmienia, zatem zostaną one jedynie opisane w kilku zdaniach. Najpierw obliczana jest delta 1, następnie tymczasowa zmienna temp. Potem jest liczona delta 2, która wraz z poprzednimi zmiennymi jest wykorzystana do obliczenia Lokatora. Po obliczeniu zmiennych pojawia się wielomian

wykrycia błędów (czyli Lokator) i następuje przejście do kolejnej części ekranu przedstawionej w kolejnym rozdziale.

#### 2.2.4.7 Szukanie pozycji błędu




Rysunek Wyliczanie wartości wielomianu w punktach

Program już wie, że we wiadomości znajduje się błąd. W tym ekranie przechodzi on do szukania, na której pozycji się on znajduje. Wykorzystuje do tego algorytm Chiena. Podobnie jak w poprzednim ekranie, zmienia się jedynie zawartość “Obszaru roboczego”, więc również będą przedstawione same zmiany we wspomnianym polu. Na początku obliczane są wartości wielomianu w kolejnych punktach do momentu, aż wartość któregoś z punktów wyniesie 0.


Rysunek Wyliczenie wartości 0


Rysunek wyliczenie pozycji błędu

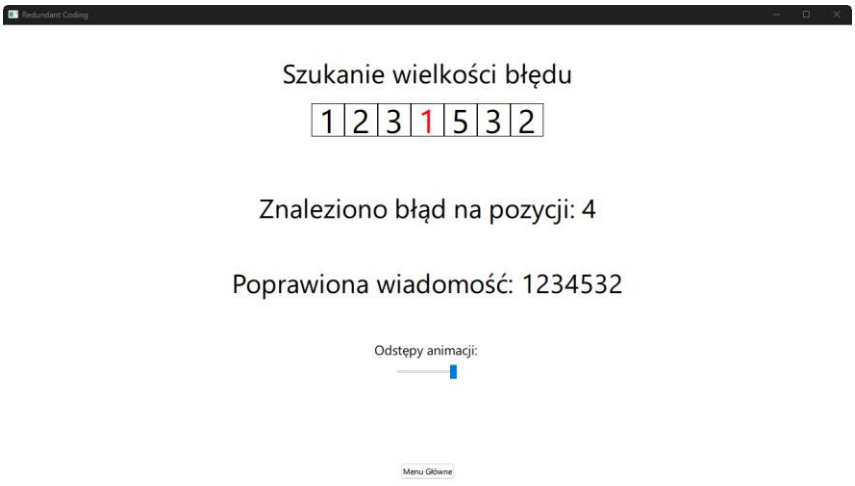

Rysunek Wynik algorytmu Chiena

Po pokazaniu przez program błędnej pozycji według algorytmu Chiena, program próbuje znaleźć prawidłową wartość w miejscu błędu, co jest przedstawione w kolejnym rozdziale.

#### 2.2.4.8 Szukanie wielkości błędu


Rysunek Pozycja błędu

Kolejną częścią algorytmu, wykonywaną w programie, jest znajdowanie prawidłowej wartości, znajdującej się na pozycji oznaczonej jako błędna. Tak jak w poprzednich rozdziałach, zawartość ekranu zmienia się jedynie w “Obszarze roboczym”, zatem tutaj również zostanie jedynie przedstawione działanie algorytmu. Najpierw zmienna coef_pos przybiera wartość pozycji wyznaczonego przez algorytm błędu. Następnie wyliczana jest errata_locator, czyli wielomian, który jest wykorzystywany do identyfikacji lokalizacji zarówno błędów, jak i korekt w przesyłanym słowie kodowym. Po tym kroku można obliczyć error_evaluator, który posłuży do obliczenia samej wartości błędu we wskazanym wcześniej przez algorytm miejscu. Dalej jest wyliczana wartość xi i y. Na koniec wyznaczana jest wielkość błędu, a po niej poprawna wiadomość.




Rysunek Ekran końcowy algorytmu. Prawidłowe wskazanie błędnej pozycji oraz poprawionej wiadomości

Użytkownik widzi na samym końcu działania algorytmu, na której pozycji program znalazł błąd oraz jaka jest według niego prawidłowa wiadomość.

## 2.3 Implementacja kodów:

Ogólna struktura implementacji obu kodowań jest podobna – istnieje klasa odpowiadająca za kodowanie zawierająca metody:

1. setInitialData – ustawiająca wstępną wiadomość, która zostanie zakodowana
2. encodeData – dokonująca kodowania wiadomości ustawionej w *setInitailData*
3. sendCode – ustawiająca otrzymaną wiadomość (może być identyczna do tej zwóconej przez *encodeData*, lub zmieniona – symulując błędy w transmisji)
4. correctError – wyszukująca błędy w wiadomości i poprawiająca je

Metody *encodeData* oraz *correctError* mają w parameterach flagę forQML – w celu odróżnienia wywołania przez frontend i przez testy. Dla testów wystarczy wykonać kodowania/dekodowanie, a w wersji dla frontendu, dodatkowo wysyłane są komunikaty informujące frontend o wykonywanych czynnościach w celu pokazania ich użytkownikowi.

Wykorzystany do tego jest mechanizm z Qt, gdzie w pliku .cpp można użyć składni *emit nazwaFunkcji*, a w .qml trzeba zaimplementować funkcję *onNazwaFunkcji*.

Po każdym komunikacie, backend czeka przez czas ustawiony w sliderze w GUI, a następnie wznawia pracę.

Obliczenia dla frontendu muszą wywoływane asynchronicznie, na nowym wątku, do czego jest wykorzystany mechanizm QtConcurrent::run. W przypadku zamknięcia okna, program w tle informuje o tym wątek i oczekuje na zakończenie działania, żeby w systemie nie został proces zombie.

### 2.3.1 Hamming

Implementacja kodu Hamminga znajduje się w pliku src/HammingCode.cpp, a nagłówek w include/HammingCode.hpp. Kod Hamminga obługuje 2 wersje – z dodatkowym bitem parzystości lub bez niego. Algorytm jest zaimplementowany używając macierzy kontroli parzystości, w której reprezentacja dziesiętna syndromu wskazuje na pozycję, na której wystąpił błąd, czyli przykładowo dla kodu Hamming(7,4)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |

Syndrom 011 w reprezentacji dziesiętnej to 3, czyli błąd znajduje się na pozycji 3 (licząc od 1). Dzięki temu nie ma potrzeby generowania i pamiętania macierzy kontroli parzystości błędów i macierzy generacyjnej.

Bity parzystości są ustawione na pozycjach będących potęgami 2 (1, 2, 4, …). Pozostałe bity są danymi, przykładowo:

| Pozycja | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| p – bit parzystości, d – bit danych | p1 | p2 | d1 | p3 | d2 | d3 | d4 |

Kolejne bity parzystości można wyliczyć jako XOR bitów na tych pozycjach, które mają „1” w reprezentacji binarnej na najmłodszym bicie (p1), drugim najmłodszym bicie (p2) itp.

Przykład dla p1:

| Pozycja | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Reprezentacja binarna | 00**1** | 01**0** | 01**1** | 10**0** | 10**1** | 11**0** | 11**1** |
| Oznaczenie | p1 | p2 | d1 | p3 | d2 | d3 | d4 |
| Czy użyć w XOR? | ✓ | | ✓ | | ✓ | | ✓ |

Przykład dla p2:

| Pozycja | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Reprezentacja binarna | 0**0**1 | 0**1**0 | 0**1**1 | 1**0**0 | 1**0**1 | 1**1**0 | 1**1**1 |
| Oznaczenie | p1 | p2 | d1 | p3 | d2 | d3 | d4 |
| Czy użyć w XOR? | | ✓ | ✓ | | | ✓ | ✓ |

Przy dekodowaniu sprawdzanie są XORy w analogiczny sposób, wynik XOR dla p1 jest najmłodszym bitem syndromu, czyli jeśliby wyszły wyniki p1=1, p2=1, p3=0, to znaczy że syndrom to 011 i jak wcześniej było wspomniane, oznacza to wykryty błąd na pozycji 3 słowie kodowym. Ten błąd jest naprawiany i następnie można odkodować wiadomość poprzez wyciągnięcie tylko bitów danych z poprawionego słowa kodowego

W przypadku dodatkowego bitu parzystości kodowanie wstępnie przebiega tak samo, na końcu jest tylko dodany nowy bit p0 na pierwszej pozycji, który zawiera XOR wszystkich bitów. Przy dekodowaniu, jeśli:

- Reprezentacja dziesiętna syndromu (C) = 0, XOR wszystkich bitów (P) = 1; wystąpił błąd w dodatkowym bicie
- C = 0, P = 0; nie znaleziono błędów
- C != 0, P = 1; znaleziono 1 błąd na pozycji C
- C != 0, P = 0; znaleziono 2 błędy, nie da się ich poprawić

Kodowanie:

```
void HammingCode::encodeDataAsync(){
    int n = this->m + this->p, dataPtr{};
    QBitArray dataEncoded(n);
    for(int i = 0; i < n; i++){
        bool isParity = isPowerTwo(i + 1),
             dataBit = data[dataPtr];
```

```
        if(!isParity) dataEncoded[i] = data[dataPtr]; //copying non-parity bits


        if(!isParity) dataPtr++;
    }
    int bit = 0;
    for(int i = 1; i <= n; i *= 2){ //calculating parity bits
        int xorVal = 0; //counting number of 1's for each parity bit, xor just signals even/odd count
        for(int j = i + 1; j <= n; j++){
            if(j & (1 << bit)){ //bit manipulation trick
                xorVal ^= dataEncoded[j - 1];
            }
        }
        dataEncoded[i - 1] = xorVal; //keeping even number of 1's in the code
        bit++;
    }
    else data = dataEncoded; //just copy the rest without extending the bit
}
```

Dekodowanie:

```
int HammingCode::correctErrorStandard()
{
    int n = this->m + this->p, C{}, bit{};
    for(int i = 1; i <= n; i *= 2){ //calculate parity bits and add them up with formula: p1 * 1 + p2 *
2 + p3 * 4 + ... = C
        int xorVal = 0;
        xorVal ^= receivedCode[i - 1]; //xor is same as counting 1's
        for(int j = i + 1; j <= n; j++){
            if(j & (1 << bit)){
                xorVal ^= receivedCode[j - 1];
            }
        }
        C += xorVal * i;
        bit++;
    }
    if(C == 0) return -1;
```

```
    else return C - 1;
}
```

### 2.3.2 Reed-Solomon

Implementacja kodu Reeda-Solomona znajduje się w pliku src/ReedSolomonCode.cpp, a nagłówek w include/ReedSolomonCode.hpp

Kodowanie Reeda-Solomona jest znacznie trudniejsze w zrozumieniu, więc powstało na podstawie przykładowej implementacji w Pythonie i zostało przeniesiona do języka C++. Użyta implementacja z wyjaśnieniem działania znajduje się pod adresem https://en.wikiversity.org/wiki/Reed%E2%80%93Solomon_codes_for_coders

Kodowanie używa w sobie teorię ciał skończonych (tzw. Ciał Galoisa - GF), które zostały zaimplementowane w wersji ogólnej (z dowolną liczbą elementów), ale w celach wizualizacji są używane tylko 8-elementowe, żeby dało się je łatwo pokazać jako 1 cyfra, symulująca blok danych 3-bitowych.

W tym ciele arytmetyka wygląda następująco:

Table 4.6 Arithmetic in GF($2^3$)

| | + | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 000 | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 001 | 1 | 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 |
| 010 | 2 | 2 | 3 | 0 | 1 | 6 | 7 | 4 | 5 |
| 011 | 3 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 |
| 100 | 4 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 |
| 101 | 5 | 5 | 4 | 7 | 6 | 1 | 0 | 3 | 2 |
| 110 | 6 | 6 | 7 | 4 | 5 | 2 | 3 | 0 | 1 |
| 111 | 7 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

(a) Addition

| | × | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 001 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 010 | 2 | 0 | 2 | 4 | 6 | 3 | 1 | 7 | 5 |
| 011 | 3 | 0 | 3 | 6 | 5 | 7 | 4 | 1 | 2 |
| 100 | 4 | 0 | 4 | 3 | 7 | 6 | 2 | 5 | 1 |
| 101 | 5 | 0 | 5 | 1 | 4 | 2 | 7 | 3 | 6 |
| 110 | 6 | 0 | 6 | 7 | 1 | 5 | 3 | 2 | 4 |
| 111 | 7 | 0 | 7 | 5 | 2 | 1 | 6 | 4 | 3 |

(b) Multiplication

Za te operacje odpowiada klasa GaloisField, która również znajduje się w pliku ReedSolomonCode.cpp.

Używane i zaimplantowane są też operacje na wielomianach reprezentowanych przez klasę Poly (także w ReedSolomonCode.cpp) zawierającą pola n – stopień wielomianu oraz int* coef – tablica ze współczynnikami wielomianu, zaczynając od najniższego (współczynnika wolnego).

Kodowanie odbywa się na wiadomości o 5 blokach danych i 2 blokach kontrolnych, pozwalających na poprawę 1 bloku błędnego o nieznanej pozycji (a zatem w GF(8) to może oznaczać błąd na 3 bitach). Najpierw, obliczane są 2 bloki kontrolne poprzez policzenie wielomianowego dzielenia wybranej wiadomości przez wielomian generujący. Współczynniki reszty z dzielenia są używane jako bloki kontrolne.

Poprawa błędów zaczyna się od policzenia syndromów, które są wartością otrzymanej wiadomości (w postaci wielomianu) w odpowiednich punktach. Jeśli wszystkie syndromy są równe 0, oznacza to, że nie wystąpił żaden błąd.

W przeciwnej sytuacji, wyliczany jest wielomian error_locator, służący do lokalizacji błędu za pomocą algorytmu Berlekamp-Masseya. Jest on użyty w algorytmie Chien żeby znaleźć na jakich pozycjach wystąpiły błędy transmisji.

Lokalizacje błędów są używane do obliczenia errata_locator – wielomianu wyszukującego błędy, który różni się od error_locator tym, że bierze pod uwagę wskazane pozycje błędów. Taka funkcjonalność nie została zaimplementowana w tym projekcie, ale typowo, jeżeli wie się, że na konkretnej pozycji wystąpił błąd, to można taką pozycję przekazać do algorytmu, co poprawia jego moc korekcyjną.

Errata_locator pozwala następnie na obliczeniu error_evaluator – wielomianu, który jest używany do obliczenia wielkości błędu (np. czy zamiast 2 powinno być 3, czy może 4).

Po obliczeniu wartości błędu na każdej pozycji, wiadomość można poprawić. Odkodowane słowo to pierwsze 5 współczynników poprawionej wiadomości.

### 2.4 Testy:

Aplikacja zawiera testy pozwalające na sprawdzenie, czy wprowadzane zmiany nie spowodowały błędów w implementacji algorytmu kodowania. Testy wykonywane są automatycznie przy uruchomieniu aplikacji w trakcie dewelopmentu i programista zostanie poinformowany, jeśli któryś z testów nie działa. Pliki z testami znajdują się w katalogu tests. W celu zmiany tego katalogu lub dodania nowego należy zmodyfikować plik CMakeLists.txt w głównym katalogu – obecnie jest tam

```
add_subdirectory(tests)
```

A w tym folderze znajduje się kolejny plik CMakeLists.txt:

```
file(GLOB FOLDER_FILES RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} *.cpp)

target_sources(appRedundantCoding PRIVATE
   ${FOLDER_FILES}
)
```

Wystarczy, że zmienna FOLDER_FILES wskazuje na wszystkie pliki z testami.

Same testy korzystają z bliblioteki GTest.

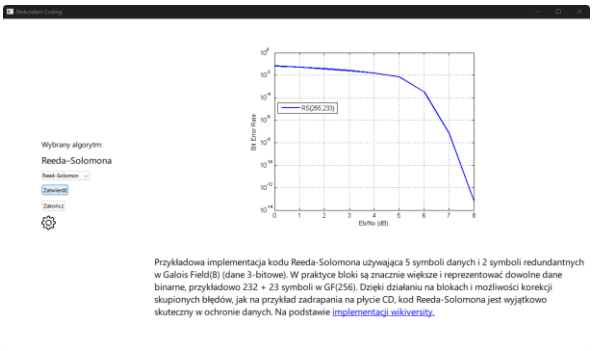
Przykładowy test:

```
////hamming's encoding testing without additional parity bit

TEST(Hamming, hammingEncoding){


    //data, expected encoded data

    QList<QPair<QString, QString>> dataAndResult{{"1110", "0010110"},
{"11010101100111011111010110", "11111010010110011111011111010110"}, {"11100001000",
"001111010001000"},

                                        {"00101010110", "110101001010110"},
{"01111000100000001001010001", "11001111100010000000001001010001"},
{"11100001011010010001110100", "11111101000101100100100001110100"},


{"01100001001000111011010010000110100100111011100100011110 1",
"01001101000100100001110110100100000110100100111011100100011110 1"},
{"11101101110000101010110100000010110110000100010010110110 0",
"00101100110111010001010101101000000010110110000100010010110110 0"},


{"10000111110010110000001001001100011001110001010111111000110111011111010100100
01000000100001101011110110101000000100010010",
"10110001011111010101100000010010001100011001110001010111111000101011101111101
01001001000000100001101011110110101000000100010010"},


{"11010001100011001111110100000101110100000100100000101000100100111001101111011
10001011100100100010100000010110101110011100011100000001011011011110110000100
00010010111101101101110111111011111001010101010001110100110000101000010011011
0111000101110100",
"01111010000110010110011111101001000101110100000100100000101000110010011100111
01111011000101110010010001010000001011010111001110010111000000010110110111101
10000100000100101111011011011101111110111110010101010100011101001100001010000
10011011001110001011101 00"}};


    foreach(auto pair, dataAndResult){


        QString data = pair.first, expectedResult = pair.second, result{};


        QBitArray bits(data.size());


        for(int i = 0; i < data.size(); i++) bits[i] = (data[i] == '1');


        auto hammingCode = QSharedPointer<HammingCode>(new HammingCode());
```

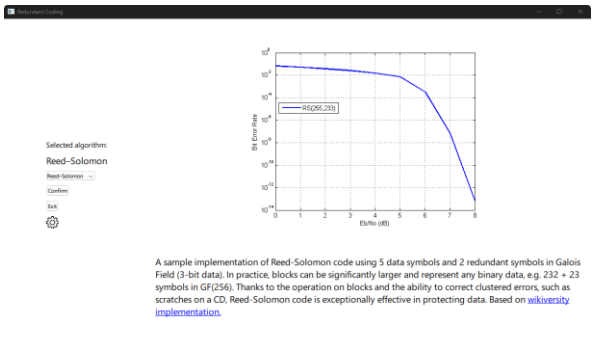
```
        hammingCode.data()->setInitialData(bits, false);
        hammingCode.data()->encodeData(false);
        QBitArray encoded = hammingCode.data()->getData();
        for(int i = 0; i < encoded.size(); i++){
            result.append(encoded.testBit(i) ? '1' : '0');
        }
        ASSERT_STREQ(result.toStdString().c_str(), expectedResult.toStdString().c_str());
    }
}
```

Każdy test powinien się zaczynać od bloku *Test(arg1, arg2) {* argumenty to kolejno nazwa zestawu testowego i nazwa konkretnego testu. W takim bloku można pisać zwykły kod C++ oraz metody ASSERT z GTest. W tym przypadku test ma zakodowane kilka możliwych inputów oraz ich poprawnie zakodowane wiadomości i dla każdej z tych par najpierw próbuje zakodować input korzystając z naszego kodu (klasa HammingCode, metoda encodeData) i następnie sprawdza, czy uzyskany wynik się zgadza z tym poprawnym (ASSERT_STREQ – porównuje, czy 2 stringi są identyczne).

## 2.5 Tłumaczenie aplikacji



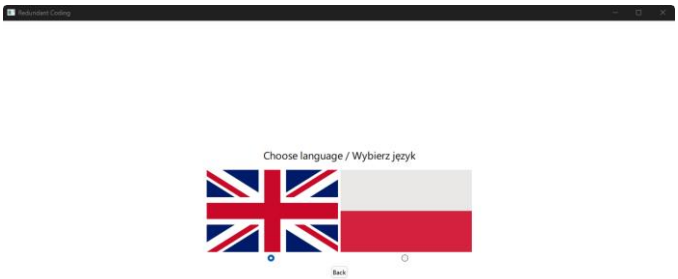

Rysunek Menu główne w języku polskim




Rysunek Menu główne w języku angielskim

Możliwa jest obsługa aplikacji w dwóch językach - polskim i angielskim. W kodzie źródłowym znajduje się instrukcja, w jaki sposób krok po kroku można dodać w razie potrzeby ewentualne dodatkowe języki. Ta opcja będzie również poniekąd przedstawiona w tym rozdziale, gdyż będzie przedstawiony sposób działania pakietów językowych w programie.

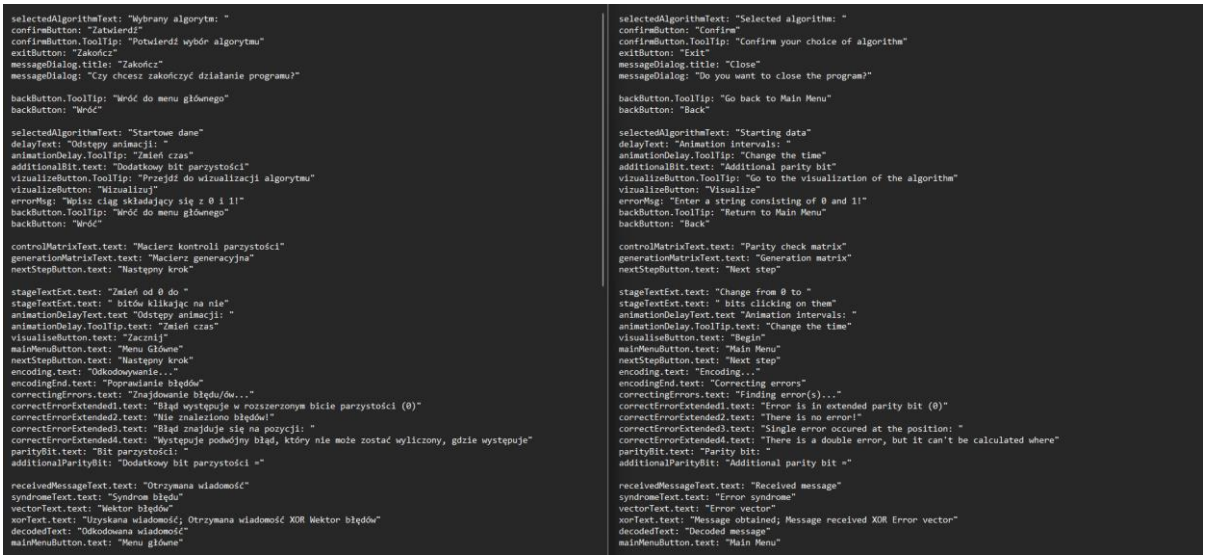
### 2.5.1 Wybór języka




Rysunek Ekran opcji

W programie język można wybrać w ekranie “Opcje”, oznaczonym w menu głównym ikonką zębatki. Użytkownik ma do wyboru język angielski, oznaczony flagą Wielkiej Brytanii oraz język polski, oznaczony flagą Polski. Pod flagami znajdują się odpowiednie przyciski typu radio. Zmiana języka następuje w chwili wciśnięcia odpowiedniego przycisku pod flagą. Jest ona od razu zapisywana do pliku konfiguracyjnego. Dzięki temu zabiegowi aplikacja działa w odpowiednim języku nawet po ponownym uruchomieniu aplikacji, więc nie ma potrzeby jego zmiany za każdym razem. Plik konfiguracyjny programu zawierający informację o języku jest zapisany w systemie Microsoft Windows w rejestrze kluczy pod ścieżką: *\HKEY_CURRENT_USER\Software\Redundant Coding\App*. Zmienna przechowująca informację o ustawionym języku programu nazywa się *Language*. Gdy jej wartość jest równa 0, to program działa w języku angielskim, gdy natomiast zmienna *Language* jest ustawiona na 1, to aplikacja jest uruchomiona w języku polskim.

### 2.5.2 Pakiety językowe



```
selectedAlgorithmText: "Wybrany algorytm: "
confirmButton: "Zatwierdź"
confirmButton.ToolTip: "Potwierdź wybór algorytmu"
exitButton: "Zakończ"
messageDialog.title: "Zakończ"
messageDialog: "Czy chcesz zakończyć działanie programu?"

backButton.ToolTip: "Wróć do menu głównego"
backButton: "Wróć"

selectedAlgorithmText: "Startowe dane"
delayText: "Odstępy animacji: "
animationDelay.ToolTip: "Zmień czas"
additionalBit.text: "Dodatkowy bit parzystości"
vizualizeButton.ToolTip: "Przejdź do wizualizacji algorytmu"
vizualizeButton: "Wizualizuj"
errorMsg: "Wpisz ciąg składający się z 0 i 1!"
backButton.ToolTip: "Wróć do menu głównego"
backButton: "Wróć"

controlMatrixText.text: "Macierz kontroli parzystości"
generationMatrixText.text: "Macierz generacyjna"
nextStepButton.text: "Następny krok"

stageTextExt.text: "Zmień od 0 do "
stageTextExt.text: " bitów klikając na nie"
animationDelayText.text "Odstępy animacji: "
animationDelay.ToolTip.text: "Zmień czas"
visualiseButton.text: "Zacznij"
mainMenuButton.text: "Menu Główne"
nextStepButton.text: "Następny krok"
encoding.text: "Odkodowywanie..."
encodingEnd.text: "Poprawianie błędów"
correctingErrors.text: "Znajdowanie błędu/ów..."
correctErrorExtended1.text: "Błąd występuje w rozszerzonym bicie parzystości (0)"
correctErrorExtended2.text: "Nie znaleziono błędów!"
correctErrorExtended3.text: "Błąd znajduje się na pozycji: "
correctErrorExtended4.text: "Występuje podwójny błąd, który nie może zostać wyliczony, gdzie występuje"
parityBit.text: "Bit parzystości: "
additionalParityBit: "Dodatkowy bit parzystości ="

receivedMessageText.text: "Otrzymana wiadomość"
syndromeText.text: "Syndrom błędu"
vectorText.text: "Wektor błędów"
xorText.text: "Uzyskana wiadomość; Otrzymana wiadomość XOR Wektor błędów"
decodedText: "Odkodowana wiadomość"
mainMenuButton.text: "Menu główne"
```

```
selectedAlgorithmText: "Selected algorithm: "
confirmButton: "Confirm"
confirmButton.ToolTip: "Confirm your choice of algorithm"
exitButton: "Exit"
messageDialog.title: "Close"
messageDialog: "Do you want to close the program?"

backButton.ToolTip: "Go back to Main Menu"
backButton: "Back"

selectedAlgorithmText: "Starting data"
delayText: "Animation intervals: "
animationDelay.ToolTip: "Change the time"
additionalBit.text: "Additional parity bit"
vizualizeButton.ToolTip: "Go to the visualization of the algorithm"
vizualizeButton: "Visualize"
errorMsg: "Enter a string consisting of 0 and 1!"
backButton.ToolTip: "Return to Main Menu"
backButton: "Back"

controlMatrixText.text: "Parity check matrix"
generationMatrixText.text: "Generation matrix"
nextStepButton.text: "Next step"

stageTextExt.text: "Change from 0 to "
stageTextExt.text: " bits clicking on them"
animationDelayText.text "Animation intervals: "
animationDelay.ToolTip.text: "Change the time"
visualiseButton.text: "Begin"
mainMenuButton.text: "Main Menu"
nextStepButton.text: "Next step"
encoding.text: "Encoding..."
encodingEnd.text: "Correcting errors"
correctingErrors.text: "Finding error(s)..."
correctErrorExtended1.text: "Error is in extended parity bit (0)"
correctErrorExtended2.text: "There is no error!"
correctErrorExtended3.text: "Single error occured at the position: "
correctErrorExtended4.text: "There is a double error, but it can't be calculated where"
parityBit.text: "Parity bit: "
additionalParityBit: "Additional parity bit ="

receivedMessageText.text: "Received message"
syndromeText.text: "Error syndrome"
vectorText.text: "Error vector"
xorText.text: "Message obtained; Message received XOR Error vector"
decodedText: "Decoded message"
mainMenuButton.text: "Main Menu"
```

Rysunek Widoczne pakiety językowe - polski i angielski

W programie używane są pakiety językowe, czyli pliki zawierające tłumaczenie każdego elementu programu. Pliki te znajdują się w folderze */assets/lang/*. Każdy z plików jest w formacie *.lang*. W pliku tym przetłumaczona jest po kolei zawartość każdej ze stron. Ważne jest by zapisywać je w odpowiedniej kolejności i w takim samym formacie, co jest widoczne na rysunku powyżej (Rysunek). Format tłumaczenia: *zmienna: “tłumaczenie”*. Format ten przypomina słownik, czyli typ danych w popularnych językach programowania, umożliwiający przypisanie do tekstu jego znaczenia. W tłumaczeniu nie może znajdować się znak cudzysłowia (“), co będzie wytłumaczone w kolejnym rozdziale, który zawiera szczegółowe omówienie tego, w jaki sposób działa tłumaczenie elementów w aplikacji.

### 2.5.3 Działanie tłumaczenia w aplikacji

Aby tłumaczenie mogło być zaimplementowane w programie, powstała osobna klasa o nazwie *Settings*, która umożliwia przechowywanie informacji odnośnie ustawień programu. Jest ona zapisana w plikach *Settings.cpp* i *Settings.hpp*. Klasa ta w ostatecznej formie umożliwia odczytywanie *getLanguage()* i zapisywanie s*etLanguage(int język)* wartości ustawionego języka w pliku konfiguracyjnym oraz odczytywanie z pakietów językowych odpowiednich tłumaczeń elementów widocznych na poszczególnych ekranach r*eadFile(int strona)*.
Kluczową spośród trzech funkcji przed chwilą wspomnianych w poprzednim akapicie jest *readFile(int strona)*. Działa ona na takiej zasadzie, że jest wywoływana za każdym razem przed przejściem do kolejnego ekranu. W pliku instruktażowym *Readme.md*, zawartym wraz z pakietami językowymi, przedstawione są numery ekranów, które należy podać jako argument *int strona*, aby uzyskać ich tłumaczenie. W plikach .qml poszczególnych ekranów, przygotowane są odpowiednie funkcje, które wstawiają tłumaczenie w elementy wskazanego ekranu. Funkcje te są automatycznie wywoływane, gdy na poprzednim ekranie użyje się funkcji *readFile(strona)* z klasy *Settings*. Ważne jest, aby nie pomylić numeru wywoływanego ekranu, gdyż inaczej doprowadzi to do nieokreślonego zachowania programu, w najlepszym przypadku dojdzie do jedynie błędnego tłumaczenia elmentów, lecz w najgorszym przypadku program przestanie działać.
Gdyby zagłębić się w szczegóły działania tłumaczenia od samego początku, wygląda to następująco:

- Z pliku .qml ekranu X jest wywołana funkcja *Settings::readFile(y)*, gdzie y to liczba określająca ekran Y
- Funkcja *readFile* znajdująca się w *Settings.cpp* wywołuje funkcję *getLanguage()* z klasy *Settings*
- Funkcja *getLanguage()* pobiera informacje z pliku konfiguracyjnego (w przypadku systemu Microsoft Windows z rejestru kluczy) dotyczącą tego, jaki język jest ustawiony
- Na podstawie uzyskanej z funkcji *getLanguage()* wartości języka, *readFile(y)* otwiera adekwatny pakiet językowy
- Po prawidłowym otwarciu pakietu językowego, funkcja czyta tekst linijka po linijce
- Wartość *y* podana jako argument do funkcji *readFile(y)* informuje ją o tym, ile znaków nowej linii funkcja musi przeczytać, zanim zacznie zapisywać tekst z pliku
- Gdy już funkcja napotkała odpowiednią liczbę znaków nowej linii, zaczyna pobierać tłumaczenie z odczytanej linii tekstu poprzez wyodrębnienie jedynie tekstu będącego pomiędzy cudzysłowiami (stąd też w tłumaczonym tekście nie może być zawartych cudzysłowiów)
- Wyodrębnione tłumaczenie jest dodawane do zmiennej output typu QString, po którym dodawany jest znak nowej linii “\n”
- Jeśli funkcja napotka na pusty znak, bądź też na koniec pliku, kończy czytać z pliku tekst i zwraca zmienną przechowującą tłumaczenie.
- Funkcja *readFile(y)* przy zwracaniu niepustej zmiennej emituje sygnał, który umożliwi ustawienie tłumaczenia na prawidłowej stronie przy jej ładowaniu
- W pliku .qml danej strony jest napisana funkcja, która otrzymuje z wyemitowanego sygnału zmienną *output*, która przechowuje w sobie tłumaczenie
- Funkcja ta rozdziela poszczególne tłumaczenia elementów, które są odseparowane znakiem nowej linii i zapisuje je do tablicy QStringów
- Na podstawie ustalonej kolejności w tejże tablicy przydzielane są tłumaczenia dla każdego z elementów

# 3 Załączniki

{wszelkie dokumenty nie dające się wkomponować w prosty sposób w tekst, należy dołączyć w osobnych plikach, a ich spis przedstawić w formie tabeli, przykładowo:}