



**Politechnika
Śląska**

PROJEKT INŻYNIERSKI

Symulator Maszyny Szyfrującej Lorenza

Jakub KUROWSKI

Nr albumu 290575

Kierunek: Teleinformatyka

PROWADZĄCY PRACĘ

Dr inż. Ewa Płuciennik

KATEDRA INFORMATYKI STOSOWANEJ

Wydział Automatyki, Elektroniki i Informatyki

GLIWICE 2023

Tytuł pracy:

Symulator Maszyny Szyfrującej Lorenza

Streszczenie:

Przedmiotem pracy było stworzenie prostego programu symulującego działanie maszyny szyfrującej Lorenza, wykorzystywanej podczas IIWS do szyfrowania transmisji telegraficznych między Naczelnym Dowództwem III Rzeszy (OKW), a sztabami armii niemieckich w okupowanej Europie. Symulator miał na celu odtworzenie podstawowej funkcjonalności maszyny – generowanie odpowiedniego klucza szyfrującego i poprawne szyfrowanie nim tekstu pobieranego przez program od użytkownika. Program został stworzony w środowisku IDE Microsoft Visual Studio 2019 Community przy pomocy języka C++, i funkcjonuje poprzez interfejs tekstowy, w oknie konsoli.

Słowa kluczowe:

Program, C++, Symulator, Interfejs tekstowy, Kryptografia, Szyfr Vernama, Maszyna Lorenza,

Thesis title:

Lorenz Cipher Machine Simulator

Abstract:

The subject of the thesis was the creation of a simple program that simulates the functioning of a Lorenz cipher machine, as utilized during WWII to cipher telegraphic transmissions between the High Command of the 3rd Reich (OKW) and the various Army Commands throughout occupied Europe. The simulator's goals were to reproduce the basic functionality of the machine – the generation of a proper encryption key and proper encryption of a text provided to the program by the user. The program was created in the Microsoft Visual Studio 2019 Community IDE with the C++ language, and functions through a text interface, in a console window.

Keywords:

Program, C++, Simulator, Text interface, Cryptography, Vernam Cipher, Lorenz Machine

Spis treści

Rozdział 1	Wstęp.....	1
Rozdział 2	Analiza tematu.....	2
Rozdział 3	Wymagania i narzędzia	8
Rozdział 4	Specyfikacja zewnętrzna programu.....	11
Rozdział 5	Specyfikacja wewnętrzna programu	15
Rozdział 6	Weryfikacja i walidacja.....	27
Rozdział 7	Podsumowanie i wnioski.....	31
Bibliografia.....		32
Spis skrótów i symboli		34
Lista dodatkowych plików, uzupełniających tekst pracy		35
Spis rysunków		36
Spis tablic		37
Spis fragmentów kodu.....		38

Rozdział 1

Wstęp

Niewątpliwie można stwierdzić, że rozwój informatyki był niemalże od samego początku ściśle związany z potrzebami militarnymi państw. Wśród wielu przykładów takiej zależności, Colossus, powszechnie uważany za pierwszy, programowalny, elektroniczny i cyfrowy komputer, został stworzony na potrzeby sekcji kryptoanalitycznej wywiadu Brytyjskiego podczas IIWS, aby zautomatyzować i ułatwić łamanie szyfrów wykorzystywanych przez armie III Rzeszy[1]. Jednakże, istnienie Colossusa nie byłoby faktem, gdyby nie wcześniejsza egzystencja maszyny szyfrującej, do którego łamania ten komputer został bezpośrednio zaprojektowany – Maszyny Szyfrującej Lorenza, potocznie nazywaną „Tuńczykiem” (z ang. *Tunny*). Gdyby nie jej istnienie, i potrzeba aliantów do deszyfrowania wiadomości generowanych przez nią, rozwój komputerów mógł przebiec inaczej, lub przynajmniej być opóźniony w czasie, gdyż jak powiada przysłowie, „potrzeba jest matką wynalazków”[2].

Celem pracy było stworzenie prostego, programowego symulatora maszyny szyfrującej Lorenza, jako swego rodzaju hołdu dla urządzenia, które do pewnego stopnia stanowi swego rodzaju punkt zapalny rozwoju komputerów – od wspomnianego wcześniej, gigantycznego Colossusa, do komputerów klasy PC, a kończąc na różnych, współczesnych mikrokontrolerach, czy też systemach na czipie (SoC, z angielskiego *System on a Chip*).

Poniższa praca zawiera, poza poniższym wstępem, sześć rozdziałów. Rozdział drugi zawiera analizę tematu, w której opisana jest historia, dzieje i działanie maszyny szyfrującej Lorenza, a także przegląd możliwych sposobów realizacji projektu symulatora. Rozdział trzeci zawiera postawione wobec tworzonego programu wymagania. Kolejne dwa rozdziały stanowią już specyfikację programu – w rozdziale czwartym zawarta jest specyfikacja zewnętrzna, czyli z perspektywy obsługi programu przez użytkownika, zaś w rozdziale piątym znajduje się dokumentacja wewnętrzna, szczegółowy opis działania i funkcjonalności stworzonego oprogramowania. Pracę kończą rozdziały szósty i siódmy, kolejno sprawozdanie z weryfikacji i walidacji działania programu, i wnioski końcowe, jednocześnie zawierające wstępne wytyczne do ewentualnego, przyszłego rozwoju programu.

Rozdział 2

Analiza tematu

Maszyna Lorenza w pigulce

Maszyna szyfrująca Lorenza, to zbiorowe określenie trzech modeli maszyn szyfrujących – SZ40, SZ42a i SZ42b - zaprojektowanych i wyprodukowanych przez niemiecką firmę C. Lorenz AG z siedzibą w Berlinie. Skrót SZ w nazwach modeli oznaczał *Schlüssel-Zusatz*, czyli „przystawka szyfrująca”, co jednocześnie w pełni opisuje formę i funkcjonalność tego sprzętu kryptograficznego – maszyny SZ były bezpośrednio podłączane do dalekopisów wykorzystywanych w centrach dowodzenia armii niemieckich, i bezpośrednio szyfrowały transmisje telegraficzne między nimi [3]. Jest to fundamentalna różnica od bardziej znanej Enigmy, która była maszyną przenośną, i wymagała obsługi przez dwie osoby – jednej generującej szyfrowany tekst, a drugiej zapisującej ręcznie wygenerowany tekst.

Biorąc pod uwagę wagę funkcji maszyn SZ – szyfrowanie transmisji między sztabami armii, co wiąże się dużym znaczeniem strategicznym przesyłanych w taki sposób meldunków – można poprosić się o myśl, że maszyny szyfrujące Lorenza musiały zapewniać bardzo skomplikowany algorytm szyfrowania. W praktyce jednak, działanie tych maszyn było jednym z lepszych przykładów, że najlepszym rozwiązaniem jest przeważnie to najprostsze.

Maszyny Lorenza wykorzystywały szyfr Vernama, nazwany na cześć jego twórcy, Gilberta Vernama, inżyniera badawczego zatrudnionego w Bell Labs, który wymyślił szyfr oparty na działaniu boolowskim XOR, znanym także jako suma modulo 2 [4][5]. Dla różnych wartości binarnych A i B, wynik ich sumy modulo 2, można opisać przy pomocy tablicy prawdy takiej jak Tabela 2.1.

Tabela 2.1 – Tablica prawdy dla $A \oplus B$

Wejście		$A \oplus B$
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

Jak można zauważyć, wynik tej operacji jest równy 1 wtedy, i tylko wtedy, jeżeli wartości A i B są sobie nierówne. Szyfr Vernama należy do rodziny algorytmów z kluczem symetrycznym, co oznacza, że ten sam klucz szyfrujący może być wykorzystany do zaszyfrowania jawnego tekstu i do deszyfrowania tekstu zaszyfrowanego tymże kluczem, czyli w naszym przypadku

$$\text{Tekst zaszyfrowany} = \text{Tekst jawny} \oplus \text{Klucz} \quad (1)$$

Oraz:

$$\text{Tekst jawny} = \text{Tekst zaszyfrowany} \oplus \text{Klucz} \quad (2)$$

W oryginalnym założeniu Vernama, szyfrowanie miało odbywać się przy pomocy dwóch taśm papierowych – jednej zawierającą tekst jawny, a drugą zawierającą jednorazowy klucz szyfrujący. Problemem w takiej realizacji jest logistyka praktycznego generowania i dystrybucji taśm kodujących, dlatego maszyny implementujące to szyfrowanie – w tym maszyny SZ Lorenza – wykorzystywały obracające się rotory do bezpośredniej generacji klucza szyfrującego.

Biorąc powyższe fakty do wiadomości, można zauważyć że w takim razie sekretem odporności takiej maszyny szyfrującej na kryptoanalizę nie jest metoda szyfrowania tekstu, ale metoda generacji klucza, która w przypadku maszyny Lorenza, była już dość rozbudowana.

Jak wspomniano powyżej, szyfr Vernama jest oparty na operacji XOR, tak więc idealnie, wygenerowany klucz szyfrujący powinien mieć tyle samo bitów, ile posiada znak tekstu do zaszyfrowania. Dalekopisy, do których podłączane były maszyny Lorenza, wykorzystywały kodowanie znaków znane jako ITA2 (z ang. *International Telegraph Alphabet No. 2*), gdzie znaki były kodowane pięciobitowo, identyczną długość miał więc generowany klucz. Strumień klucza generowany przez maszynę miał charakterystykę pseudolosową, i był oparty, jak wspomniano o tym wcześniej, na obracających się rotorach. Maszyna Lorenza zawierała 12 rotorów, dwie grupy po pięć rotorów szyfrujących (*Chi* i *Psi*) oraz 2 rotory kontrolne (*mu*). Każdy z rotorów zawierał pewną liczbę pinów, które mogły być skonfigurowane na stan wysoki (logiczna 1) lub niski (logiczne 0), co wpływało na wartość generowanego klucza. Dodatkowo, aby zmaksymalizować czas przed powtórzeniem się sekwencji, wszystkie koła miały względnie pierwszą (czyli bez

wspólnych dzielników większych od 1) liczbę pozycji (pinów) na których się mogły zatrzymać. Pełen rozkład pozycji/pinów kół maszyny jest zaprezentowany w Tablicy 2.2:

Tablica 2.2 – Liczby pinów/pozycji na kołach maszyny Lorenza

Nazwa koła	<i>Psi 1</i>	<i>Psi 2</i>	<i>Psi 3</i>	<i>Psi 4</i>	<i>Psi 5</i>	<i>Mu 1</i>	<i>Mu 2</i>	<i>Chi 1</i>	<i>Chi 2</i>	<i>Chi 3</i>	<i>Chi 4</i>	<i>Chi 5</i>
Liczba pinów	43	47	51	53	59	37	61	41	31	29	26	23

Czas między powtórzeniami sekwencji kół, jest iloczynem ilości ich pozycji, dla kół *Chi* wynosi on 22041682 dla kół *Psi* 322303017, zaś ilość kombinacji wszystkich kół – *Chi*, *Psi*, *Mu* – wynosi $1,603 \cdot 10^{19}$.

Generacja klucza polegała na pobraniu przez maszynę pięciobitowej wartości z kół *Chi* i pięciobitowej wartości z kół *Psi*, i wykonanie operacji sumy modulo 2 między nimi. Tak stworzony klucz końcowy był następnie wykorzystywany do szyfrowania podanego na wejście znaku, będącego częścią tekstu. Z każdym kolejnym znakiem, koła szyfrowe *Chi* obracały się o jedną pozycję do przodu, natomiast obrót kół szyfrowych *Psi* był oparty na obrotach dwóch kół *Mu*. Koło *Mu 1* obracało się wraz z kołami *Chi*, zaś koło *Mu 2*, tylko wtedy, gdy pin na *Mu 1* miał logiczną jedynkę. Kiedy zaś sam *Mu 2* wystawiał logiczną jedynkę, dopiero wówczas koła *Psi* mogły się obracać. Aby zapewnić jeszcze większą odporność maszyny na kryptoanalizę, ustawienia pinów na rotorach były regularnie zmieniane.

Złamanie szyfru

Opisany wcześniej sposób generacji klucza niewątpliwie sugeruje wysoki poziom trudności w złamaniu szyfru. Jednakże, jak w każdym innym systemie nawet do dziś, jego bezpieczeństwo jest tak dobre, jak jego najsłabsze ogniwo – w tym przypadku, człowiek, będący operatorem maszyny. Wywiad brytyjski miał pojęcie o istnieniu maszyn szyfrujących Lorenza od połowy 1940 roku, gdyż wówczas angielskie stacje nasłuchowe, które dotychczas przechwytywały transmisje kodem Morsa – w szczególności, stacja na Denmark Hill w Camberwell w Londynie – zaczęły odbierać nowy typ transmisji telegraficznych, jednakże ze względu na brak zasobów operacyjnych, analiza tych nowych transmisji miała niski priorytet. Dopiero po ustanowieniu nowej stacji nasłuchowej w Knockholt w hrabstwie Kent (południowo-wschodnia Anglia) na początku 1941, której celem głównym było właśnie przechwytywanie i ewentualne przekazywanie dla potrzeb

kryptoanalitycznych do centrum kryptograficznego w Bletchley Park tychże nowych transmisji^[6]. I tak, 30 sierpnia 1941, wiadomość transmitowana z Aten do Wiednia nie została odebrana poprawnie, co spowodowało wysłanie z Wiednia do Aten zwrotnego, niezaszyfrowanego zapytania o retransmisję, która została wykonana przy szyfrowaniu tym samym kluczem co oryginał – praktyka stanowiąca błąd krytyczny obsługi jakiegokolwiek szyfru strumieniowego, zwłaszcza że retransmitowany tekst był streszczeniem oryginalnego, błędnie przesłanego tekstu. Takie zajście pozwoliło wywiadowi brytyjskiemu monitorującemu szyfrowane niemieckie transmisje telegraficzne (które Brytyjczycy nazywali „fish” czyli ryba, dlatego też maszyny SZ były nazywane „tunny”, czyli tuńczyk) na uzyskanie dwóch związanych ze sobą tekstów szyfrowych, co rozpoczęło proces łamania działania maszyny Lorenza. Na samym początku, wywiadowi brytyjskiemu udało się uzyskać oryginalną treść przechwyconych meldunków oraz ich strumień kluczowych, jednakże trudność sprawiła dedukcja sposobu generacji klucza. Po trzech miesiącach, prace nad rozwiązaniem tejże zagadki przekazano matematykowi Williamowi Thomasowi Tutte, któremu udało się rozpoznać powtarzalność strumienia kluczowego, co mocno przyspieszyło kryptoanalizę maszyny^[7]. Do Stycznia 1942, w zajściu nazwanym „jednym z największych intelektualnych osiągnięć Drugiej Wojny Światowej”^[2], brytyjskim kryptoanalitykom udało się w pełni odwzorować strukturę logiczną działania maszyny Lorenza i od tamtej pory, zaczął się proces deszyfrowania przechwytywanych wiadomości, które ze względu na ich wysoką wagę strategiczną, przyczyniły się do szybszego zwycięstwa sił Aliantów w Europie.

Deszyfrowanie i rozwój komputerów

Po złamaniu szyfru, kolejnym krokiem było zapewnienie deszyfrowania przechwytywanych meldunków – zadanie te zostało oddelegowane zespołowi pod dowództwem Ralpha Testera, od którego nazwiska zespół ten był nazywany potocznie *Testery*. Jednakże, deszyfrowanie wiadomości było zbyt żmudne i czasochłonne aby pracę wykonywał tylko człowiek, dlatego też zespół Testera był wspomagany maszynami operowanymi przez zespół pod dowództwem Maxa Newmana, nazywanego potocznie *Newmanry*^[8]. Pierwszą maszyną wykorzystywaną do łamania szyfrów był tzw. *British Tunny*, czyli funkcjonalna kopia maszyny Lorenza, oparta na jej strukturze logicznej, zrekonstruowanej przez Tutte’a i jego współpracowników^[9]. Jednakże, do jej funkcjonowania wymagana była znajomość ustawień kół szyfrujących, które początkowo były uzyskiwane dzięki maszynom zaprojektowanym dla *Newmanry* przez Heath Robinsona, wykorzystujące dwie taśmy papierowe i obwody logiczne do dedukcji ustawień kół *Chi* maszyny Lorenza. Były one jednak problematyczne –

zwłaszcza w przypadku synchronizacji między taśmami – i relatywnie zbyt powolne jak na potrzeby personelu kryptograficznego w Bletchley Park [10]. Prawdziwym przełomem okazał się *Colossus*, zaprojektowany przez Thomasa Harolda Flowersa, i będący w zamyśle znacznie szybszy, wydajniejszy i mniej awaryjny niż maszyny Robinsona. Warto zaznaczyć, że pomysł Flowersa był początkowo traktowany sceptycznie przez jego przełożonych, którzy chcieli się skoncentrować na dalszej eksploatacji maszyn Robinsona, jednakże mimo tego Flowers postanowił kontynuować swój projekt, pokrywając część kosztów ze swojej kieszeni – do takiego stopnia, że pod koniec wojny, był on już zadłużony [11]. Jednakże, ten wysiłek nie wyszedł na marne – w pełni operacyjny już Colossus był naprawdę dużo szybszy i wydajniejszy (np. Robinsony odczytywały 2000 znaków na sekundę, Colossus już 5000) oraz wymagał tylko jednej taśmy, zamiast dwóch, co jednocześnie eliminowało problemy synchronizacyjne, i był także programowalny przy pomocy tablicy programowej (plugboard) oraz kabli rozruchowych[10]. Do końca wojny uruchomiono, aż dziesięć tych maszyn, z czego pierwsza była w pełni operacyjna już w grudniu 1943.

Symulowanie maszyny Lorenza

Z funkcjonalnego punktu widzenia, szyfrowanie maszyny Lorenza jest proste w realizacji programowej – kod znaku należy podać operacji XOR wygenerowanego klucza. Oczywiście, współczesne komputery nie kodują znaków w standardzie ITA2 – najprostszy tryb znakowy dostępny w językach programowania to przeważnie *char*, reprezentujący znaki kodowane przy pomocy standardu ASCII (z ang. *American Standard Code for Information Interchange*). W takiej sytuacji są dwa różne rozwiązania:

- Proste, naiwne rozwiązanie, gdzie ignorujemy kodowanie ITA2 i szyfrujemy bezpośrednio znaki ASCII. Problemem jest fakt, że klucz szyfrujący maszyny Lorenza jest pięciobitowy, a znaki ASCII są kodowane ośmiobitowo.
- Trudniejsze, ale bardziej adekwatne dla programu symulatora, gdzie następuje emulacja ITA2 – użytkownik podaje znak ASCII, program pobiera skojarzony z danym znakiem kod ITA2, który jest szyfrowany kluczem, i zwraca znak ASCII skojarzony z otrzymanym kodem zaszyfrowanego znaku ITA2.

Na potrzeby bardziej realistycznej symulacji, zdecydowano się na opcję numer dwa.

Kolejnym krokiem jest niewątpliwie wybór interfejsu użytkownika, przez który następuje interakcja między nim, a programem wirtualnej maszyny szyfrującej. Ponownie, są dwie opcje:

- Interfejs graficzny, potencjalnie łatwiejszy i bardziej przejrzysty w obsłudze, ale wymagający większego nakładu pracy i wymagań sprzętowych do uruchomienia programu
- Interfejs tekstowy, uruchamiany z okna konsoli systemu operacyjnego, łatwiejszy w realizacji, ale pozostawiający sobie wiele do życzenia z punktu widzenia użytkownika.

Na potrzeby projektu zdecydowano się na stworzenie programu z interfejsem tekstowym, gdyż uznano że ważniejsze jest oddanie serca działania maszyny Lorenza – sposobu jej szyfrowania, generowania klucza itp. – niż wymyślna szata graficzna programu której realizacja może być czasochłonna z minimalnymi benefitami funkcjonalnymi.

Ostateczną decyzją był wybór języka programowania w którym można było stworzyć program. Wybór jest potencjalnie ogromny – od Java po Python i wiele innych - ale ostatecznie zdecydowano się na programowanie w języku C++, głównie ze względu na wcześniejsze doświadczenie w pracy z tym językiem programowania.

Rozdział 3

Wymagania i narzędzia

Wymagania funkcjonalne i нефункционалне

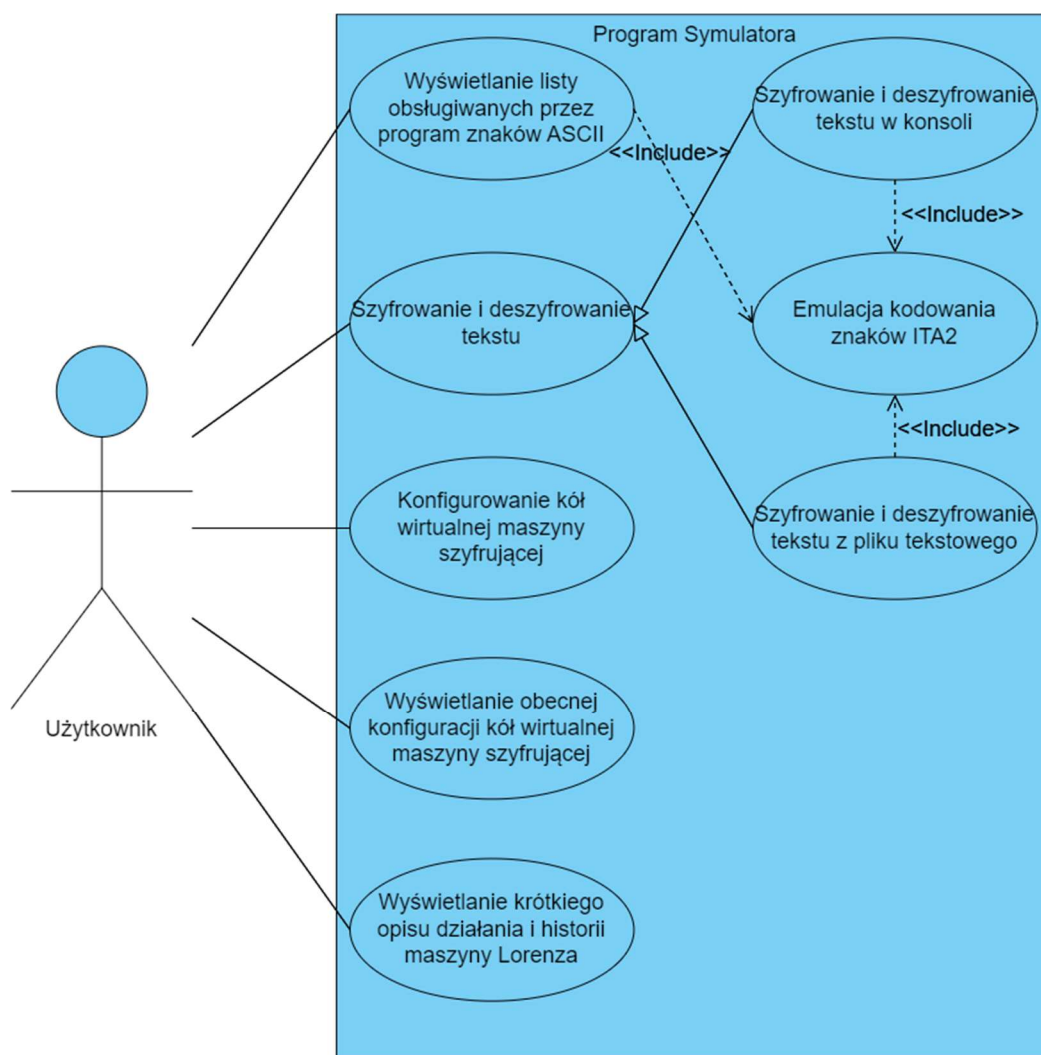
Za wymagania funkcjonalne programu uznano:

- Zdolność programu do przetworzenia podanego przez użytkownika tekstu, identycznie do działania oryginalnej maszyny szyfrującej. Użytkownik powinien mieć możliwość podania tekstu albo bezpośrednio, w oknie konsoli, lub poprzez podanie odpowiedniego pliku tekstowego, które program powinien móc odczytać. Wynik przetwarzania tekstu powinien być generowany bezpośrednio po jego zakończeniu – w przypadku tekstu pobranego bezpośrednio, w oknie konsoli w której funkcjonuje program, zaś w przypadku odczytu z pliku program powinien generować plik wyjściowy, zawierający tekst wyjściowy
- Użytkownik powinien mieć możliwość skonfigurowania wirtualnych kół szyfrujących symulowanej maszyny Lorenza, oraz możliwość sprawdzenia ich obecnej konfiguracji
- Ze względu na potrzebę emulowania kodowania ITA2 w programie symulatora, i co za tym idzie inny zbiór obsługiwanych znaków, niż typowy dla przeciętnego użytkownika standard ASCII czy też Unicode, program powinien umożliwić użytkownikowi na wgląd do obsługiwanego zbioru znaków, w celu zapewnienia poprawnej obsługi programu
- Ze względu na częściowy charakter edukacyjny symulatora, wyświetlanie krótkiej informacji o maszynie Lorenza powinno być sensownym dodatkiem do powyższych, krytycznych z punktu widzenia symulacyjnego, funkcjonalności

Za wymagania niefunkcjonalne uznano:

- Interfejs użytkownika, będący interfejsem tekstowym, powinien być jak najprostszy w obsłudze – ilość wymaganych od użytkownika działań powinna być sprowadzona do minimum
- Wydajność – program ten nie powinien do uruchomienia wymagać zasobów systemowych znacznie odchodzących od jego natury. Program uruchamiany w oknie konsoli systemu operacyjnego nie powinien być równie wymagający zasobowa, jak pełnoprawna aplikacja z interfejsem graficznym.
- Prostota – jakakolwiek funkcjonalność odchodząca od wcześniej wytyczonych wymagań funkcjonalnych nie powinna być realizowana, kod powinien być prosty i w miarę możliwości jak najbardziej przejrzysty

Diagram przypadków użycia, oparty na ww. wymaganiach został zaprezentowany na rysunku 3.1:



Rys.3.1 – Diagram przypadków użycia programu symulatora

Narzędzia

Do stworzenia programu przy pomocy języka programowania C++ użyto środowiska IDE Microsoft Visual Studio 2019 Community. Wybór ten został zmotywowany wcześniejszą znajomością i doświadczeniem w korzystaniu z powyższego oprogramowania. Dodatkowo, do modyfikowania plików tekstowych używanych podczas testowania działania programu wykorzystano Notatnik systemu Windows- prosty edytor tekstu, o dostatecznej funkcjonalności dla tej potrzeby.

Rozdział 4

Specyfikacja zewnętrzna programu

Wymagania sprzętowe i programowe

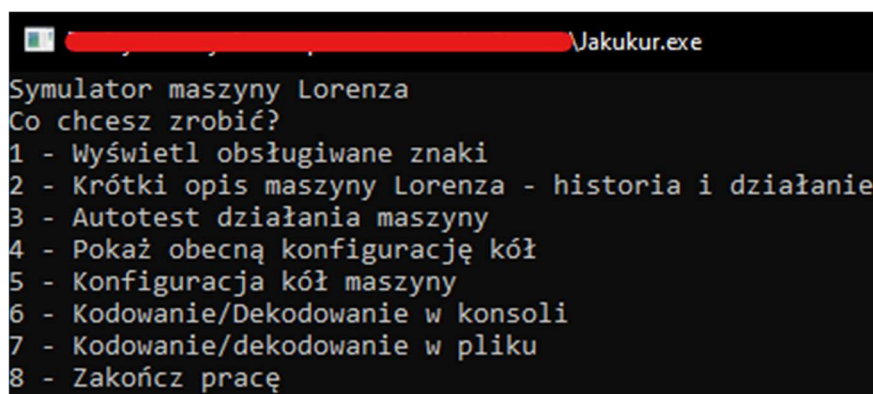
Ponieważ program działa w oknie konsoli systemu Windows, program ten nie posiada żadnych specyficznych dla siebie wymagań sprzętowych, wymagany jest tylko komputer klasy PC z zainstalowanym na nim systemem operacyjnym Windows 10.

Instalacja i aktywacja

Ze względu na to, że program działa w oknie konsoli, nie wymaga on instalacji, użytkownik musi tylko uruchomić plik wykonywalny *Jakukur.exe*.

Obsługa programu

Po uruchomieniu programu powinno wyświetlić się pokazane w Rys. 4.1 okno konsoli, na którym wyświetlone będzie menu główne programu:



```
Symulator maszyny Lorentza
Co chcesz zrobić?
1 - Wyświetl obsługiwane znaki
2 - Krótki opis maszyny Lorentza - historia i działanie
3 - Autotest działania maszyny
4 - Pokaż obecną konfigurację kół
5 - Konfiguracja kół maszyny
6 - Kodowanie/Dekodowanie w konsoli
7 - Kodowanie/dekodowanie w pliku
8 - Zakończ pracę
```

Rys. 4.1 – Początkowe okno programu

Ze względu na fakt, że program działa w konsoli systemu Windows, do obsługi wymagana jest klawiatura. Użycie myszki jest czysto opcjonalne, i sprowadza się do ewentualnego zaznaczania przez użytkownika tekstu wyświetlanego w oknie konsoli, na potrzeby np. skopiowania go do schowka.

Aby wybrać daną opcję podaną w menu, należy podać jej numer, poprzez wciśnięcie odpowiedniego klawisza numerycznego na klawiaturze, i potwierdzenie wyboru klawiszem *enter*. Opcje 1 – 4 tylko wyświetlają informacje dla użytkownika, zaś opcje 5 –

Rys. 4.4 – Konfiguracja kół

Wówczas wygenerowane sekwencje będą miały kształt jak przedstawiono na Rys. 4.5.:

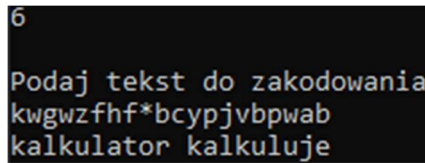
Rys. 4.5 – Wygenerowane sekwencje na kołach

Warto wspomnieć, że nowa konfiguracja kół jest ważna tylko na czas obecnej sesji w programie, po jego ponownym uruchomieniu, program wraca do konfiguracji domyślnej.

6 – Pozwala użytkownikowi na zakodowanie lub zdekodowanie tekstu w oknie konsoli. Użytkownik może tutaj podać tekst dowolnej długości, zawierający spacje, który zostanie w całości zakodowany przez program i wyświetlony w oknie konsoli. Alternatywnie, użytkownik może podać zakodowany ciąg znaków, i jeżeli obecna konfiguracja kół maszyny w programie jest taka sama jak w momencie tworzenia tekstu zakodowanego, to wówczas rezultatem będzie zdekodowany tekst jawny. Przykładowo, jeżeli zakodujemy taki sam tekst, jak przedstawiono na Rys. 4.6.:

Rys.4.6 – Ręczne kodowanie przykładowego tekstu

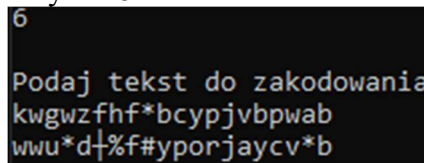
To jeśli sekwencje na kołach są takie same jak były w momencie zakodowania tekstu jawnego, to wtórne podanie tekstu zakodowanego powinno zwrócić zdekodowany tekst jawny, jak na Rys.4.7.:



```
6
Podaj tekst do zakodowania
kwgwzfhf*bcypjvbpwab
kalkulator kalkuluje
```

Rys. 4.7 – Ręczne dekodowanie tekstu zakodowanego w Rys. 4.6

Jeżeli zaś sekwencja jest inna, wówczas zamiast zdekodować, tekst zostanie zakodowany wtórnie, jak na Rys. 4.8.:



```
6
Podaj tekst do zakodowania
kwgwzfhf*bcypjvbpwab
wwu*d|f#yporjaycv*b
```

Rys. 4.8 – Podwójne kodowanie tekstu

Powyższe zachowanie jest zgodne z sposobem kodowania tekstu przez maszynę Lorenza, ale z punktu widzenia użytkownika jest działaniem niechcianym, dlatego ważne jest zapewnienie przez użytkownika zgodności między sekwencjami kół użytych do kodowania i dekodowania.

7 – Pozwala użytkownikowi na odczyt pliku tekstowego i jego zakodowanie (lub zdekodowanie, jeżeli plik zawiera tekst zakodowany tą samą sekwencją na kołach jak ta, którą jest obecnie skonfigurowany program). Użytkownik podaje ścieżkę do pliku tekstowego wraz z jego rozszerzeniem (najlepiej *.txt*), program odczytuje plik, koduje/dekoduje jego zawartość, a następnie zapisuje wynik do nowego pliku tekstowego *output.txt*, znajdującego się w tym samym katalogu co program.

8 – Kończy pracę programu.

Rozdział 5

Specyfikacja wewnętrzna programu

Założenia podstawowe

Program został napisany w języku C++ bez wykorzystywania paradygmatu programowania obiektowego. Działanie programu opiera się na pętli, w której w zależności od komendy podawanej na wejście wywoływane są odpowiednie funkcje i podfunkcje, realizujące odpowiednie elementy działania programu. Większość zmiennych wykorzystywanych w programie jest zadeklarowanych globalnie.

Wykorzystywane biblioteki

Program wykorzystuje tylko następujące domyślne biblioteki C++:

- `<iostream>`, zapewniającą obsługę pobierania od użytkownika danych i wyświetlania ich przez program w oknie konsoli
- `<string>`, pozwalającą na tworzenie i obsługę zmiennych typu *string*, dynamicznych tablic znaków
- `<map>`, pozwalającą na tworzenie i obsługę zmiennych typu *map*, sortowanych list par klucz-wartość
- `<fstream>`, zapewniającą obsługę odczytu z i zapisu do plików przez program
- `<locale>`, pozwalającą na zmianę *locale* programu, co pozwala na wyświetlanie przez program polskich znaków w konsoli.

Dodatkowo, w programie jest domyślnie wykorzystywana przestrzeń nazw (*namespace*) *std*. Jednakże nie jest to funkcjonalność krytyczna do działania programu, przestrzeń została wykorzystana dla ułatwienia procesu tworzenia kodu.

Zmienne programu

Program wykorzystuje następujące typy zmiennych:

- Zmienne dynamicznych tablic znaków typu *string*.
- Zmienne liczb całkowitych *int*.
- Zmienne strumienia dostępu do plików *ifstream* (strumień odczytu) i *ofstream* (strumień zapisu)
- Zmienna sortowanej listy par klucz-wartość typu *map*, sortowana według kluczy komórek.

Zmienne dynamicznych tablic znaków *string*:

- Zmienne *chi1*, *chi2*, *chi3*, *chi4*, *chi5*, *mu1*, *mu2*, *psi1*, *psi2*, *psi3*, *psi4*, *psi5* służą do przechowywania binarnych sekwencji stanów na wirtualnych kołach kodowych symulowanej przez program maszyny szyfrującej Lorenza.
- Zmienne *input*, *ouput* są wykorzystywane w procesie kodowania/dekodowania tekstu, gdzie zmienna *input* zawiera tekst do przetworzenia, który po przetworzeniu jest zapisywany do zmiennej *output*.
- Zmienna *buffer*, jak nazwa sugeruje, jest wykorzystywana jako bufor odczytu, podczas operacji na pliku tekstowym

Zmienne liczb całkowitych *int*:

- Zmienne *ic1*, *ic2*, *ic3*, *ic4*, *ic5*, *im1*, *im2*, *ip1*, *ip2*, *ip3*, *ip4*, *ip5* służą do przechowywania wartości związanych z wirtualnymi kołami szyfrującymi. Rodzaj tych wartości zależy od obecnego stanu maszyny – podczas kodowania tekstu, zmienne te zawierają obecne wartości 0 lub 1 znajdujących się na kołach szyfrujących, zaś w przypadku konfiguracji kół, zmienne te wówczas przechowują rozmiary kół, jako długość odpowiedniej zmiennej *string* reprezentującą dane koło szyfrujące.
- Zmienne *i*, *j*, *k* służą jako indeksy odpowiadające obecnie pobieranym wartościom z kół szyfrujących, *i* jest wykorzystywane dla kół *chi* i *mu1*, *j* dla kół *psi*, a *k* dla koła *mu2*.
- Zmienna *codekey* zawiera obecnie obliczoną wartość klucza szyfrującego
- Zmienna *encoded* zawiera wartość obecnego zakodowanego znaku
- Zmienna *control* jest zmienną kontrolną wykorzystywaną do kontroli menu w pętli głównej programu

Zmienne strumienia dostępu do plików *ifstream* i *ofstream*.

- Zmienna *ifstream infile* służy do dostępu do pliku wejściowego, z którego program pobiera dane
- Zmienna *ofstream outfile* służy do dostępu do pliku wyjściowego, do którego program zapisuje dane

Zmienna sortowanej listy par klucz-wartość *map*.

Zmienna *map<char,int> ITA2mod* służy do swoistej symulacji kodowania znaków ITA2, na którym były oparte oryginalne, fizyczne maszyny szyfrujące Lorenza. W każdej komórce tej mapy znajduje się para klucz – wartość, gdzie klucz, jako typ znakowy *char*, odpowiada znakowi kodowanego standardem komputerowym ASCII, a wartość liczby całkowitej typu *int* skojarzona z danym kluczem jest wartością kodu ITA2 odpowiadającemu danemu znakowi. Warto zaznaczyć, że zmienna *ITA2mod*, nie reprezentuje kodowania ITA2 w stosunku 1:1 – jest to spowodowane zastąpieniem

kodowania znaków kontrolnych dodatkowymi znakami specjalnymi. Taka realizacja została spowodowana chęcią zapewnienia większej stabilności podczas procesu odczytu i kodowania tekstu. Dodatkowo, ze względu na to że ITA2 nie rozróżnia między dużymi a małymi literami, zmienna *mapy* zawiera tylko małe litery, plus cyfry i wybrane znaki specjalne, oraz znak spacji – w sumie *ITA2mod* przechowuje 64 różnych par.

Opis funkcji

Funkcjonalnie program podzielony jest na dwa typy funkcji, *główne* i *pomocnicze*.

Funkcje główne są bezpośrednio wywoływane w pętli głównej programu poprzez instrukcję *switch* i poza trzema wyjątkami zawierają wywołania funkcji pomocniczych i dodatkowe, proste działania, takie jak wyświetlanie tekstu czy też czyszczenie zmiennych. Celem tych funkcji jest kontrola wykonywania danej funkcjonalności programu, która jest realizowana przez odpowiednie wywoływanie funkcji pomocniczych.

Funkcje pomocnicze służą do wykonywania powtarzalnych działań przez funkcje nadrzędne, ich istnienie pozwala na lepszą organizację kodu podczas jego tworzenia – dla przykładu, jakiegokolwiek problemy z daną funkcjonalnością programu łatwiej naprawić modyfikując funkcje pomocnicze wywoływane przez funkcję główną, niż modyfikować jedną, bardzo rozwiniętą funkcję. Jak wcześniej wspomniano, są wywoływane przez inne funkcje, w niektórych przypadkach więcej niż raz.

Pętla główna programu

Pętla główna programu jest zawarta w funkcji *int main()*, której większość należy do pętli *while*. Przebieg działania jest następujący:

- *locale* programu jest ustawiany na obsługę polskich znaków i formatów dat i liczb, poprzez funkcję biblioteczną *setlocale(LC_ALL, „pl_PL”)*.
- Program wyświetla tekst *Symulator maszyny Lorenza*
- Program wchodzi w pętlę *while*. Pętla ta będzie działać w nieskończoność dopóki wartość globalnej zmiennej kontrolnej *int control* jest różna od 8 (domyślnie, wartość zmiennej wynosi 0). Przebieg pętli *while* jest następujący:
 - Wykonywana jest instrukcja *switch*, która w zależności od obecnej wartości zmiennej *control* wywołuje odpowiednie funkcje. Reaguje on na wartości *control* będącymi liczbami całkowitymi od 1 do 7. Przy pierwszej iteracji pętli, która odbywa się po uruchomieniu programu, *control* jest równy wartości domyślnej 0, dlatego *switch* nie wykona się.
 - Wywoływana jest funkcja *void attract()*, która wyświetla w oknie konsoli tekst menu.
 - Program czeka na podanie przez użytkownika wartości zmiennej *control*. Podanie wartości liczby całkowitej z zakresu od 1 do 7

sprawi, że następna iteracja pętli wywoła odpowiedni *case* instrukcji *switch*. Podanie wartości 8 powoduje spełnienie warunku wykonywania pętli *while*, a co za tym idzie zakończy zarówno pętlę, jak i działanie programu, gdyż pętla *while* jest ostatnim elementem wykonywanym w *int main()*. Podanie innych wartości całkowitych spowoduje ponowne wykonanie się pętli *while* tak jak w przypadku pierwotnego uruchomienia programu – pojawia się ponownie menu. Jeżeli jednak użytkownik poda wartość niezgodną z typem zmiennych *int* jaką jest zmienna *control* – np. literę, znak specjalny, ciąg znaków, czy też liczbę niecałkowitą – wówczas następuje działanie odwrotne od zamierzonego i program zapętla się, i wymagany jest wówczas jego restart.

Funkcje główne

Funkcje główne są podawane zgodnie z kolejnością z jaką są osadzone w pętli głównej programu, wraz z podaniem wywoływanych przez nie funkcji pomocniczych, a także wartości *control* wywołujących je.

- **void texthelp(map<char, int>input)**
 - Wywoływana dla *control* o wartości 1.
 - Nie wywołuje żadnych funkcji pomocniczych
 - Jedna z trzech funkcji głównych, które nie wywołują funkcji pomocniczych, i które można byłoby skategoryzować za pomocnicze, gdyby nie fakt, że są bezpośrednio wywoływane przez użytkownika poprzez instrukcję *switch* w pętli głównej programu.
 - Funkcja ta ma za zadanie wyświetlenie obsługiwanych przez program znaków, które mogą być poprawnie zakodowane i zdekodowane, innymi słowy, wyświetla ona zawartość zmiennej *map<char,int> ITA2mod*. Funkcja iteruje kolejno po komórkach mapy i wyświetla skojarzoną z daną komórką wartość klucza. Dla potrzeb estetycznych, znaki są wyświetlane w liczbie 14 znaków na linię
- **void lorenzinfo()**
 - Wywoływana dla *control* o wartości 2.
 - Nie wywołuje żadnych funkcji pomocniczych
 - Kolejna funkcja, którą można byłoby skategoryzować za pomocniczą. Służy tylko i wyłącznie do wyświetlania krótkiej informacji tekstowej na temat historii i działania maszyny szyfrującej Lorenza.
- **void test()**
 - Wywoływana dla *control* o wartości 3.
 - Wywołuje funkcję pomocniczą *void encode(string toencode)*.

- Pierwsza funkcja główna wywołująca funkcje pomocnicze, *void test()* jak nazwa sugeruje, służy do szybkiego testu funkcjonalności symulatora przy obecnej konfiguracji kół szyfrujących. Funkcja zapisuje do zmiennej *string input* prosty tekst, który jest następnie przekazywany do podfunkcji *encode*, która koduje podany ciąg znaków, który jest zapisywany w zmiennej *string output*. Następnie, *test* wyświetla w oknie konsoli oryginalny tekst wraz z jego wersją po zakodowaniu. Kolejnym krokiem jest tekst dekodowania – do zmiennej *input* przypisywana jest zawartość zmiennej *output*, która jest zaś czyszczona z obecnej zawartości przy pomocy funkcji bibliotecznej *.clear()* właściwej dla biblioteki *<string>*. Następnie znowu *input* jest przekazywany do *encode*, i jeżeli wszystko powiodło się pomyślnie, zwrócona nam nowa zawartość *output*, która zostanie wyświetlona w oknie konsoli powinna być równoważna z oryginalnym tekstem testowym. Na sam koniec działania funkcji *test*, zmienne *input* i *output* są opróżniane z zawartości, znów przy pomocy zmiennej bibliotecznej *.clear()*.
- **void wheelinfo()**
 - Wywoływana dla *control* o wartości 4.
 - Nie wywołuje żadnych funkcji pomocniczych.
 - Ostatnia z funkcji quasi-pomocniczych, *wheelinfo()* wyświetla obecne sekwencje na wirtualnych kołach szyfrujących symulowanej maszyny szyfrującej Lorenza. Ze względu na fakt, że sekwencje na kołach są przechowywane w programie jako zmienne *string*, działanie tej funkcji sprowadza się do wyświetlania tychże zmiennych.
- **void configwheels()**
 - Wywoływana dla *control* o wartości 5.
 - Wywołuje funkcję pomocniczą *string wheelconfig(string wheel, int wheelsize, string configseq)*.
 - Funkcja służąca do konfiguracji sekwencji na wirtualnych kołach szyfrujących, co pozwala na stworzenie nowego szyfru. Dla każdego koła, po kolei funkcja przypisuje do odpowiedniej dla niego zmiennej iteracyjnej jego rozmiar (dla przykładu, dla koła Chi 1, jego długość jest zapisywana do zmiennej iteracyjnej *ic1*), następnie prosi użytkownika o podanie sekwencji konfiguracyjnej, która jest zapisywana w zmiennej *string input*. Sekwencja ta może zawierać dowolne znaki ASCII, zarówno alfanumeryczne jak i specjalne. Następnie, funkcja wywołuje podfunkcję *wheelconfig*, której argumentami wywoławczymi jest *string* obecnie konfigurowanego koła, odpowiedni iterator typu *int*, zawierający długość koła, a także sekwencję konfiguracyjną, również

jako *string*. Po skonfigurowaniu jednego koła, funkcja przechodzi do konfigurowania kolejnego - znów zapisuje jego długość, pobiera od użytkownika sekwencję konfiguracyjną i wywołuje podfunkcję. Proces ten powtórzy się dla wszystkich wirtualnych kół szyfrujących maszyny – w sumie dwanaście razy. **Użytkownik musi ręcznie skonfigurować wszystkie koła, gdyż funkcja *configwheels* nie przewiduje przedwczesnego przerwania jej działania.**

- **void writeandcode()**

- Wywoływana dla *control* o wartości 6.
- Wywołuje funkcję pomocniczą *void encode(string toencode)*.
- Funkcja pozwalająca na szybkie zakodowanie/zdekodowanie wiadomości bezpośrednio w oknie konsoli. Funkcja pobiera od użytkownika ciąg znaków zapisywany do zmiennej *input*, które są w zamyśle obsługiwane przez program, przy pomocy podfunkcji *encode* tworzy na podstawie ciągu znaków i obecnej kombinacji sekwencji na kołach szyfrujących tekst wyjściowy zapisywany do zmiennej *output*, a następnie wyświetla wynik działania w oknie konsoli. Podawany przez użytkownika ciąg znaków jest pobierany przy pomocy funkcji bibliotecznej *getline*. Użycie *getline* pozwala na pobieranie tekstu zawierającego odstępy między znakami i wyrazami, co pozwala użytkownikowi na zakodowanie całych zdań, a nie tylko pojedynczych wyrazów. Przed wywołaniem *getline* funkcja też zawiera wywołanie *cin.ignore()*, gdyż inaczej, ze względu na to że funkcja *writeandcode* wykonuje się w pętli *while*, użytkownik nie mógłby podać jakichkolwiek danych, a program mógłby się zapętlić. Po wyświetleniu tekstu wynikowego, program kasuje obecną zawartość zmiennych *input* i *output*, poprzez funkcję biblioteczną biblioteki *<string>* *.clear()*.

- **void filecode()**

- Wywoływana dla *control* o wartości 7.
- Wywołuje funkcję pomocniczą *void encode(string toencode)*.
- Funkcja zapewniająca funkcjonalność kodowania i dekodowania plików tekstowych. Na samym początku użytkownik podaje nazwę lub ścieżkę do pliku tekstowego, wraz jego rozszerzeniem, którego zawartość chce zakodować/zdekodować. Program zakłada, że użytkownik będzie próbował dokonać odczytu prostego pliku tekstowego typu *.txt*, zawierającego tylko i wyłącznie znaki obsługiwane przez program. Podobnie jak w funkcji *writeandcode*, ciąg znaków podawany przez użytkownika jest pobierany poprzez *getline* do zmiennej *input*. Następnie zmienna *ifstream* dokonuje próby otwarcia pliku tekstowego

o nazwie/ścieżce podanej jako *input*, z włączonymi flagami odczytu *ios::in* i *ios::binary* (innymi słowy, jest to operacja *infile.open(input, ios::in | ios::binary)*). Jeżeli otwarcie nie powiedzie się sukcesem, program wyświetla informację o błędzie odczytu i prośbie od użytkownika, o sprawdzenie, czy poprawnie podał nazwę i/lub pełną ścieżkę dostępu do pliku. Jeżeli zaś otwarcie pliku tekstowego powiedzie się sukcesem, wówczas program odczytuje zawartość pliku. Program najpierw przygotowuje zmienną *string buffer*, w której zapisana zostanie zawartość pliku – przy pomocy funkcji bibliotecznej biblioteki *<fstream>* *infile.seekg(0, ios::end)* szuka końcowej pozycji w pliku tekstowym, a następnie wykonuje *buffer.resize(infile.tellg())*. Operacja ta sprowadza się do ustawienia rozmiaru zmiennej *string buffer* na wartość równą ostatniej pozycji w pliku tekstowym – innymi słowy, *buffer* przyjmuje jako swój rozmiar ilość znaków w pliku tekstowym. Po dokonaniu tej operacji, wracamy na „początek” pliku poprzez wywołanie funkcji bibliotecznej *infile.seekg(0, ios::beg)*, po czym wykonujemy odczyt właściwy, poprzez wywołanie innej funkcji bibliotecznej – tym razem jest to *infile.read(&buffer[0], buffer.size())*. Pierwszy argument tej funkcji oznacza tablicę, do której będą zapisywane dane, zaś drugi argument, oznacza ilość znaków które odczytujemy – w tym przypadku, oznacza to, że dla *string buffer* o danym sobie rozmiarze, który wcześniej został ustawiony na ilość znaków w pliku tekstowym, tyle znaków ile może on pomieścić zostanie do niego wpisanych. Po wykonaniu odczytu, program zamyka dostęp do pliku tekstowego, poprzez funkcję biblioteczną *infile.close()*. Po zamknięciu pliku, program w końcu koduje zawartość *buffer* przy pomocy podfunkcji *encode*, która jest zapisywana do zmiennej *output*. Następnie, tworzony i otwierany jest plik tekstowy, poprzez operację na zmiennej *ofstream outfile* przy pomocy funkcji bibliotecznej – *outfile.open(„output.txt”)*, gdzie *output.txt* to nazwa tworzonego pliku tekstowego, zawierającego wynik kodowania tekstu pobranego z pliku tekstowego. Program zapisuje zawartość *output* do obecnie otwartego *outfile* przy pomocy zwykłego operatora strumienia (*outfile << output*), a na samym końcu zamyka dostęp do pliku wyjściowego (*outfile.close()*) i kasuje zawartość wszystkich użytych dotychczas zmiennych typu *string* – *input*, *buffer*, *output* – przy pomocy funkcji bibliotecznej *.clear()*.

Funkcje pomocnicze

Funkcje pomocnicze są podawane w kolejności od najmniej do najbardziej rozbudowanych, dodatkowo podane jest gdzie te funkcje są wywoływane i czy same z siebie nie wywołują innych podfunkcji:

- **int char2num (char inchar)**

- Wywoływana przez funkcję *void encode(string toencode)*.
- Prosta funkcja, mająca na celu konwersję znaku cyfry w kodowaniu ASCII na wartość całkowitą przez nią reprezentowaną, co sprowadza się to pobrania znaku, i wykonania prostego działania:

$$\text{Wartość liczbowa} = \text{Kod ASCII cyfry} - 48 \quad (3)$$

Powyższy wynik jest zwracany przez funkcję. Teoretycznie, powyższa funkcja może wykonywać działanie odwrotne od zamierzonego, jeżeli na wejście nie zostanie podany znak cyfry, jednakże ze względu na sposób wywoływania tej funkcji sprawia, że nieprawidłowe działanie tej funkcji jest niemożliwe podczas normalnego funkcjonowania programu.

- **char num2char(int innum)**

- Wywoływana przez funkcję *string wheelconfig(string wheel, int wheelsize, string configseq)*.
- Kolejna prosta funkcja, o działaniu odwrotnym do *int char2num(char inchar)*, funkcja ta wykonuje następujące działanie:

$$\text{Kod ASCII} = \text{Wartość liczbowa} + 48 \quad (4)$$

W zamyśle docelowym, funkcja ta ma za zadanie przekonwertować wartość cyfry 0 - 9 na odpowiadający jej znak ASCII, w innych przypadkach, które podobnie jak w przypadku *char2num* są niemożliwe podczas poprawnego działania programu, funkcja zwracałaby jakiś znak ASCII kodowany na wartości która została obliczona.

- **char find_mapkey(map<char,int>inmap, int value)**

- Wywoływana przez funkcję *void encode(string toencode)*
- Funkcja służąca do wyszukiwania w mapie typów *<char, int>* przekazanej jako *inmap* klucza skojarzonego dla wartości *value*. Funkcja iteruje po kolejnych komórkach *inmap* od początku do końca, sprawdzając wartości drugiego pola komórek, jeżeli taka wartość jest równa wartości *value*, to funkcja zwraca klucz dla danej komórki. Teoretycznie, *inmap* mogłaby zawierać kilka komórek, których wartości drugiego pola komórki są sobie równe, pod warunkiem, że ich klucze są unikatowe. Wówczas funkcja zwracałaby klucz, którego komórka jako pierwsza spełniła warunek wyszukiwania, gdyż w mapie, komórki są sortowane względem kluczy. W programie takie zajście jest

niemożliwe, gdyż jedyna wykorzystywana w nim mapa ma zarówno unikatowe klucze jak i unikatowe wartości komórek

- **int calc_codekey(int c1,int c2,int c3,int c4,int c5,int p1,int p2,int p3,int p4,int p5)**

- Wywoływana przez funkcję *void encode(string toencode)*.
- Funkcja generująca klucz szyfrujący, na podstawie podanych wejściu wartości na kołach Chi i Psi wirtualnej maszyny szyfrującej Lorenza. Funkcja najpierw składa wartości kół na dwie liczby pięciobitowe – efekt jest osiągany poprzez odpowiednie lewostronne przesunięcia bitowe wartości na kolejnych kołach. Przy założeniu, że koła o najmniejszym indeksie (Chi 1 i Psi 1) posiadają wartości najbardziej znaczących bitów liczb pięciobitowych, działanie tej funkcji można zapisać wzorem

$$\sum_{n=1}^5 a(n) * 2^{5-n} \quad (5)$$

gdzie n to indeks koła danego typu, zaś a to wartość binarna na kole. Złożone w taki sposób liczby pięciobitowe są zapisywane lokalnie wewnątrz funkcji jako *int chifull* i *int psifull*. Na sam koniec, funkcja oblicza właściwy klucz szyfrujący, według równania:

$$\text{Klucz szyfrujący} = \text{chifull} \oplus \text{psifull} \quad (6)$$

czyli operacji logicznej XOR, inaczej tzw. sumy modulo 2, której wynik jest zwracany przez funkcję.

- **string wheelconfig(string wheel, int wheelsize, string configseq)**

- Wywoływana przez funkcję *void configwheels()*.
- Wywołuje funkcję *char num2char(int in)*.
- Funkcja generująca nową sekwencję na wirtualnym kole szyfrującym, najpierw kasuje obecną zawartość zmiennej przekazanej jako *string wheel*, która odpowiada sekwencji na obecnie konfigurowanym kole szyfrującym. Następnie program sprawdza, czy sekwencja konfiguracyjna, przekazana jako *string configseq* ma długość równą rozmiarowi koła szyfrującego, przekazanego jako *int wheelsize*. Najpierw sprawdzane jest, czy sekwencja jest za krótka, jeżeli tak, to jest ona powiększana przy pomocy poniższego algorytmu:

```
if (configseq.length() < wheelsize)
{
    for (int m = configseq.length(); m < wheelsize; m += m)
    {
        configseq += configseq;
    }
}
```

Kod. 5.1 – Kod powielający sekwencję konfiguracyjną

Co sprowadza się do powielania sekwencji konfiguracyjnej, dopóki jej długość nie przekroczy rozmiaru koła szyfrującego. Kolejnym krokiem jest sprawdzanie czy sekwencja konfiguracyjna nie jest za długa – czy to poprzez powyższe powielanie, czy też tak została podana przez użytkownika. Jeżeli tak, to wówczas przy pomocy funkcji bibliotecznej `.erase` biblioteki `<string>`, kasowana jest zawartość `configseq` między pozycją o indeksie równym rozmiarowi koła, a dotychczasowym końcem `configseq`. Po wykonaniu tych dwóch sprawdzeń generowana jest sekwencja na kole szyfrującym:

```
for (int n = 0; n != wheelsize; n++)
{
    wheel += num2char(configseq[n]%2);
}
```

Kod. 5.2 – Generacja sekwencji szyfrującej

Powyższe działanie sprowadza się do sprawdzania parzystości kodów ASCII znaków będących częścią `configseq` poprzez operację modulo 2, a następnie konwersja uzyskanych wartości liczbowych 0 lub 1 na kod znaków ASCII cyfr 0 lub 1, które są dodawane na koniec `wheel`, który po zakończeniu działania funkcji zostanie przez nią zwrócony jako nowa sekwencja na danym kole szyfrującym.

- **void encode(string toencode)**
 - Wywoływana w `void filecode()`, `void writeandcode()` i `void test()`
 - Wywołuje funkcje `int char2num(char inchar)`, `int calc_codekey(int c1,int c2,int c3,int c4,int c5,int p1,int p2,int p3,int p4,int p5)`, i `char find_mapkey(map<char,int>input, int value)`
 - Funkcja stanowiąca swoiste serce symulatora, gdyż symuluje ona właściwe kodowanie znaków w podanym jako `toencode` tekście. Na samym początku funkcja resetuje do zera wartości `i,j,k`, które reprezentują indeksy obecnie pobieranych wartości bitowych z kół szyfrujących. Zmienna `i` jest też wykorzystywana jako inkrementator pętli `for`, w której ta funkcja wykonuje większość swoich działań. Na samym początku pętli sprawdzamy czy w obecnej chwili nie próbujemy zakodować kombinacji znaków kontrolnych CR LF, oznaczających rozpoczęcie pisania tekstu od nowej linii. Jeżeli tak, to program nie koduje tych znaków (gdyż nie znajdują się w `ITA2mod`), zamiast tego wpisuje na sztywno do `output` znak kontrolny LF. Sposób realizacji tego działania może sprawiać problemy z kompatybilnością –w systemie operacyjnym Windows 10, przed znakiem kontrolnym LF zawsze jest automatycznie generowany znak CR, które razem właśnie reprezentują

nową linię. W przypadku systemów uniksopodobnych (Linux, macOS itp.), nowa linia jest reprezentowana tylko znakiem kontrolnym LF, i o ile samo wpisywanie znaku kontrolnego LF byłoby zgodne, to warunek detekcji, czyli szukanie kombinacji CR LF, może powieść się niepowodzeniem. Słowem kluczowym jest może – działanie programu nie był testowany pod systemem uniksopodobnym. Kontynuując, następnym krokiem funkcji jest rotacja wirtualnych kół szyfrujących i pobieranie z nich wartości bitowych. Podobnie jak w oryginalnej maszynie szyfrującej Lorenza, koła Chi i Mu 1 obracają się co każdy kolejny znak, co w programie jest utożsamione z inkrementacją wartości i , jak wspomniano na początku, zaś ich obecne wartości są zapisywane do zmiennych $ic1$, $ic2$, $ic3$, $ic4$, $ic5$, $im1$. Koło Mu 2, sterujące obrotem kół Psi, obraca się tylko wtedy, gdy koło Mu 1 miało w poprzedniej iteracji wartość 1, co w programie jest odzwierciedlone przez następujące działanie – jeżeli pobrana wartość $im1$ jest równa 1, wówczas inkrementowana jest wartość k . Inkrementacja k następuje dopiero po pobraniu i zapisaniu do zmiennej $im2$ obecnej wartości na kole Mu 2 – zmiana wartości będzie odczuwalna dopiero w kolejnej iteracji. Na sam koniec, koła Psi obracają się tylko wtedy, jeżeli obecna wartość na kole Mu 2 jest 1 – i podobnie jak w przypadku k , w programie wspomniana rotacja jest przypisana do zmiennej j , która jest inkrementowana jeżeli pobrana wartość $im2$ jest równa 1. Z racji tego, że koła są reprezentowane poprzez zmienne typu *string*, czyli dynamiczne tablice znaków, każde pobranie obecnej wartości z koła wiąże się z wywołaniem podfunkcji *char2num*. Dodatkowo, koła szyfrujące w pewnym momencie przejdą przez pełen zakres swoich wartości i zaczną obracać się „od nowa” – programie jest to zaimplementowane, poprzez pobieranie znaku na indeksie wyznaczonym prostym równaniem:

$$\text{Indeks na kole} = (\text{Wartość inkrementatora}) \bmod (\text{Rozmiar koła}) \quad (7)$$

Po pobraniu wartości ze wszystkich kół szyfrujących, i zapisania ich do odpowiednich zmiennych, następuje wywołanie podfunkcji *calc_codekey*, która oblicza klucz szyfrujący dla obecnego znaku, zapisywany do zmiennej *int codekey*. Następnie generowany zostaje kod znaku zaszyfrowanego *int encoded* jako wynik działania

$$\text{encoded} = \text{ITA2mod}[\text{toencode}[i]] \oplus \text{codekey} \quad (8)$$

gdzie pierwsza wartość oznacza pobraną z mapy *ITA2mod* wartość symulowanego zmodyfikowanego kodowania ITA2 dla obecnie szyfrowanego znaku *toencode[i]*. Na samym końcu do zmiennej

wynikowej *string output* dodawany jest zakodowany znak, wyznaczony poprzez wywołanie podfunkcji *find_mapkey(ITA2mod, encoded)*, która dla wartości *encoded* powinna znaleźć w *ITA2mod*, znak będący kluczem skojarzonym z tą wartością. Po wykonaniu tych operacji, następuje inkrementacja *i* oraz przejście w kolejną iterację pętli *for*, która będzie się wykonywać tak długo, jak długi był ciąg znaków do zakodowania, czyli innymi słowy, dopóki $i < toencode.length$.

Rozdział 6

Weryfikacja i walidacja

Funkcjonalność i działanie programu symulatora były testowane przy każdym kroku jego realizacji. Jak było to wspomniane w *Specyfikacji wewnętrznej programu*, ostateczna wersja symulatora opisywana w poniższym dokumencie zawiera potencjalne zachowania uznawane za nieprawidłowe, jednakże są to wydarzenia z kategorii takich, które użytkownik musiałby z własnej woli wywołać, gdyż podczas normalnej pracy, zgodnej ze *Specyfikacją zewnętrzną programu* takie zajścia nie powinny się wydarzyć.

W obecnej wersji programu, użytkownik może spowodować tylko następujące, błędne zachowanie:

- W menu głównym programu, gdzie użytkownik jest proszony przez symulator do podania liczby uruchamiającej daną funkcję, domyślnie oczekiwane jest podanie przez użytkownika liczby całkowitej z zakresu 1-8, jednakże nic nie stoi na przeszkodzie, aby użytkownik podał inną wartość. Podanie wartości całkowitej nie należącej do wyżej wymienionego zakresu powoduje ponowne wyświetlenie się menu głównego, gdyż program po prostu nie otrzymał wartości, na którą miał zareagować. Jednakże, jeżeli użytkownik poda wartość innego typu – liczbę wymienną, znak lub ciąg znaków – to wówczas program wchodzi w pętlę nieskończoną, gdyż użytkownik podał na wejście wartość niezgodną ze zmienną *control*, która kontroluje działanie menu, i jest ona zmienną typu liczby całkowitej *int*. Oczywiście, takie działanie jest związane z jawnym zignorowaniem instrukcji wyświetlanych przez program w oknie konsoli.
- Podczas szyfrowania znaków, nie jest sprawdzane, czy dany znak jest obsługiwany, program po prostu oczekuje od użytkownika podawania znaków zgodne z wytycznymi podawanymi przez opcję *Wyświetl obsługiwane znaki*. Jeżeli użytkownik jednak poda znak nieobsługiwany przez program, to wówczas program dodaje nowe wartości do *map<char, int> ITA2mod*. Jest to spowodowane poniższą linijką kodu:

```
127      encoded = ITA2mod[toencode[i]] ^ codekey;
```

Kod. 6.1 – Kod szyfrowania znaku

a w szczególności fragmentem $ITA2mod[toencode[i]]$. W normalnym przypadku funkcjonowania programu, fragment ten wywołuje dostęp do komórki $ITA2mod$ indeksowanym kluczem o wartości znaku podanego jako $toencode[i]$. Jednakże, operator $[]$ zmiennej typu *map* ma też drugą funkcjonalność – jeżeli mapa nie zawiera komórki o kluczu podanym w nawiasie operatora $[]$ i „pustej” wartości – w przypadku $ITA2mod$ jest to wartość *int* równa 0, i takie zdarzenie właśnie następuje w programie. Jednakże, ze względu na specyfikę implementacji $ITA2mod$ w większości przypadków po takim zejściu program będzie dalej działał poprawnie. Aby to wytłumaczyć, trzeba dogłębnie wytłumaczyć sposób implementowania „zmodyfikowanego” kodowania ITA2 jako $ITA2mod$ w programie. Jako punkt wyjściowy, oczywiście wzięto oryginalny standard kodowania znaków ITA2, który zawiera dwie grupy znaków – literową i liczbową, przełączaną przez odpowiedni znak kontrolny:

Tabela 6.1 – Kody znaków w standardzie ITA2

ITA2 – kody znaków literowych (<i>letters set</i>)																
HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	<i>NUL</i>	E	<i>LF</i>	A	<i>SP</i>	S	I	U	<i>CR</i>	D	R	J	N	F	C	K
1x	T	Z	L	W	H	Y	P	Q	O	B	G	<i>FIGS</i>	M	X	V	<i>LTRS/DEL</i>
ITA2 – kody znaków liczbowych (<i>figure set</i>)																
HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	<i>NUL</i>	3	<i>LF</i>	-	<i>SP</i>	'	8	7	<i>CR</i>	<i>ENQ</i>	4	<i>BEL</i>	,	!	:	(
1x	5	+)	2	£	6	0	1	9	?	&	<i>FIGS</i>	.	/	=	<i>LTRS</i>

Znaki oznaczone pogrubioną kursywą są znakami kontrolnymi, wśród nich znajduje się para znaków *LTRS/FIGS*, które właśnie służą do przełączania między grupami znaków. Znaki nie będące znakami kontrolnymi zostały zaimplementowane następująco:

- Litery zostały zaimplementowane 1:1, z takimi samymi indeksami bitowymi, jak bazowy ITA2, ale w postaci małych liter, a nie dużych. Uznano, że łatwiej będzie z punktu widzenia użytkownika pisać tylko małymi literami, niż tylko dużymi literami, co wymagałoby albo przytrzymywania klawisza SHIFT podczas pisania, lub przełączenia CAPS LOCK
- Liczby i znaki specjalne zostały zaimplementowane z indeksami powiększonymi o wartość szesnastkową 0x20, odpowiadającą jedynie na 6

bicie od prawej. Decyzja była pokierowana faktem, że przełączanie między grupami znaków przy pomocy znaków kontrolnych *LTRS/FIGS*, można zinterpretować jako aktywacja i dezaktywacja dodatkowego bitu kontrolnego w kodzie znaku.

Znaki kontrolne kodowane w ITA2 zostały zastąpione wybranymi, unikatowymi znakami specjalnymi ASCII, ze względu na chęć zapewnienia stabilności odczytu tekstu i jego szyfrowania, a także fakt, że zmienna typu *map*, która została wybrana na emulację kodowania ITA2, nie dopuszcza duplikatów kluczy oraz, jak wspomniano w *Specyfikacji wewnętrznej programu*, może sprawiać problemy wyszukiwania kluczy jeżeli są w parach z identycznymi wartościami, po których ich szukamy (poza znakiem *SP*, który oznacza spację, który można było zaimplementować normalnie, i został tak zaimplementowany, jako część grupy liter, czyli z wartością szesnastkową 0x04)

Ostatecznie, *ITA2mod* reprezentuje sposób kodowania znaków podany w tabeli 6.2.

Tabela 6.2 – Kody znaków w *ITA2mod*

ITA2mod																
HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	;	e	#	a	SP	s	i	u	%	d	r	j	n	f	c	k
1x	t	z	l	w	h	y	p	q	o	b	g	^	m	x	v	*
2x	_	3	{	-	}	'	8	7	<	>	4	\	,	!	:	(
3x	5	+)	2	\$	6	0	1	9	?	&		.	/	=	„

Jak wiadomo, zmienna typu *map* jest automatycznie sortowana względem ich kluczy, które w naszym przypadku są zmiennymi typu *char*, które reprezentują znaki ASCII, dlatego więc, aby nowo wprowadzona para klucz-wartość „nadpisała” dotychczasową parę klucz-wartość, gdzie wartość jest równa 0, musi to być para o kluczu, którego kod ASCII jest mniejszy od znaku średnika (;), i który nie został zaimplementowany w *ITA2mod*. Poza znakami kontrolnymi, nie ma takiego znaku, więc „nadpisanie” klucza skojarzonego z wartością 0 jest mało prawdopodobne. We wcześniejszej wersji programu już takie zdarzenie było w miarę prawdopodobne, gdyż z wartością 0 w *ITA2mod* był skojarzony klucz odpowiadający znakowi @, który mógł być właśnie nadpisany przez średnik, dlatego obecna wersja *ITA2mod* zawiera średnik, zamiast @. Oczywiście, efektem ubocznym powyższej modyfikacji jest zamiana obsługi jednego znaku na obsługę drugiego, ale ciężko stwierdzić, czy jest to jakaś fundamentalna zmiana – jeśli chodzi o teksty jawne, obydwa znaki pojawiają się dość nieczęsto, nawet w tekstach w języku angielskim, pod

względem którego zarówno ITA2, jak i ASCII, oraz wywodzący się z nich *ITA2mod* są zaprojektowane.

Oprócz wyżej wspomnianych zachowań, wszystkie inne wykryte błędne działania programu zostały usunięte.

Rozdział 7

Podsumowanie i wnioski

Celem pracy było stworzenie prostego programu symulatora maszyny szyfrującej Lorenza. Stworzony program działa zgodnie z założeniami i zapewnia podstawową funkcjonalność wytyczoną w procesie analizy wstępnej tematu projektu. Jest jednak dość oczywiste, że pod niektórymi względami program mógłby być w przyszłości zdecydowanie rozwinięty:

- Interfejs tekstowy, o ile zapewniający dostateczną kontrolę nad symulatorem w jego obecnym kształcie, jest z punktu widzenia potencjalnego, przeciętnego użytkownika nieintuicyjny – rozbudowa programu pod kątem implementacji w pełni funkcjonalnego interfejsu graficznego z obsługą myszki byłaby dobrym punktem wyjściowym do dalszej rozbudowy symulatora
- Konfiguracja wirtualnych kół szyfrujących niewątpliwie pozostawia wiele do życzenia – w obecnym kształcie wymaga ona od użytkownika ręcznego skonfigurowania wszystkich kół po kolei, bez możliwości przerywania procesu i/lub skonfigurowania tylko pewnej części kół. Potencjalną ścieżką rozwoju mogłaby być nie tylko implementacja możliwości przerywania ręcznej konfiguracji, ale także możliwości konfiguracji kół na podstawie wczytanego pliku, zawierającego sekwencje konfiguracyjne dla poszczególnych kół
- W końcu, zapewnienie większej kompatybilności między różnymi systemami operacyjnymi niewątpliwie mogłoby zapewnić większą bazę potencjalnych użytkowników

Realizacja powyższych punktów jako wytycznych do dalszego rozwoju oprogramowania mogłaby potencjalnie przekształcić obecną wersję symulatora w pełnoprawny produkt do celów edukacyjno-rekreacyjnych i dać mu prawo bytu w przypadku ewentualnego wprowadzenia go na rynek.

Bibliografia

- [1] Dermot Turing, *XYZ. Prawdziwa Historia złamania szyfru Enigmy*, wydawnictwo Rebis, 2019
- [2] Tony Sale, *The Lorenz Cipher and how Bletchley Park broke it*, dostępne na <https://www.codesandciphers.org.uk/lorenz/fish.htm> (dostęp: 05.01.2023)
- [3] Marcin Karbowski *Podstawy Kryptografii* wydanie III, wydawnictwo Helion, 2021
- [4] Otwarte zasoby Politechniki Warszawskiej, *Podstawy Techniki Cyfrowej – Kody i Szyfry*, dostępne na <https://esezam.okno.pw.edu.pl/mod/book/view.php?id=40&chapterid=863> (dostęp: 05.01.2023)
- [5] Melville Klein, *Securing Record Communications: The TSEC/KW-6* dostępne na https://web.archive.org/web/20120315235937/http://www.nsa.gov/about/_files/crypto_logic_heritage/publications/misc/tsec_kw26.pdf (dostęp: 05.01.2023)
- [6] Jack Good, Donald Michie, Geoffrey Timms, *General Report on Tunny: With Emphasis on Statistical Methods* (1945), z archiwów UK Public Record Office (obecnie The National Archives) w Kew – dokumenty HW 25/4 i HW 25/5, kopia dostępna na http://www.alanturing.net/turing_archive/archive/index/tunnyreportindex.html (dostęp: 05.01.2023)
- [7] William Thomas Tutte, *Fish and I*, 1998, dostępne na <https://web.archive.org/web/20150212204111/https://cryptocellar.web.cern.ch/cryptocellar/tutte.pdf> (dostęp: 07.01.2023)
- [8] UCL News, *Jerry Roberts' codebreaking talk*, dostępne na <https://www.ucl.ac.uk/news/2009/mar/watch-now-jerry-roberts-codebreaking-talk> (dostęp 07.01.2023)
- [9] The Register, *Bletchley Park completes epic Tunny Machine*, dostępne na https://www.theregister.com/2011/05/26/bletchley_park_tunny_rebuild_project/ (dostęp 07.01.2023)
- [10] Jack Copeland, *Colossus: The Secrets of Bletchley Park's Codebreaking Computers*, Oxford University Press, 2006, ISBN 978-0-19-284055-4
- [11] Margaret Ann Boden, *Mind as Machine: A History of Cognitive Science*, Clarendon Press, 2006, ISBN 9780199543168

Dodatki

Spis skrótów i symboli

<i>IIWŚ</i>	II Wojna Światowa
<i>WWII</i>	II Wojna Światowa (z ang. <i>World War II</i>)
<i>OKW</i>	Naczelne Dowództwo Sił Zbrojnych (z niem. <i>Oberkommando der Wehrmacht</i>)
<i>SoC</i>	System na Czipie (z ang. <i>System on a Chip</i>)
<i>ITA2</i>	Międzynarodowy Alfabet Telegraficzny nr 2 (z ang. <i>International Telegraph Alphabet No. 2</i>)
<i>ASCII</i>	Amerykański Standardowy Kod dla Wymiany Informacji (z ang. <i>American Standard Code for Information Interchange</i>)

Lista dodatkowych plików, uzupełniających tekst pracy

W systemie, do pracy dołączono dodatkowe pliki zawierające:

- źródła programu,
- dane testowe
- film pokazujący działanie opracowanego oprogramowania

Spis rysunków

3.1	Diagram przypadków użycia programu	9
4.1	Początkowe okno programu	11
4.2	Obsługiwane znaki tekstu kodowanego lub dekodowanego	12
4.3	Wyświetlanie domyślnych sekwencji na kołach wirtualnej maszyny Lorenza	12
4.4	Konfiguracja kół	13
4.5	Wygenerowane sekwencje na kołach	13
4.6	Ręczne kodowanie przykładowego tekstu	13
4.7	Ręczne dekodowanie tekstu zakodowanego w Rys. 4.6	14
4.8	Podwójne kodowanie tekstu	14

Spis tablic

2.1	Tablica prawdy dla $A \text{ XOR } B$	3
2.2	Liczby pinów/pozycji na kołach maszyny Lorenza	4
6.1	Kody znaków w ITA2	28
6.2	Kody znaków w <i>ITA2mod</i>	29

Spis fragmentów kodu

5.1	Kod powielający sekwencję konfiguracyjną	23
5.2	Generacja sekwencji szyfrującej	24
6.1	Kod szyfrowania znaku	28