

Projektowanie elektronicznych układów sterowania

Projekt: Symulacja komputerowa obsługi silnika z
wykorzystaniem magistrali komunikacyjnej CAN

Skład grupy:

Imię	Nazwisko	Nr albumu
Maciej	Białecki	239395
Marcin	Kania	239402

Spis treści

1. Wstęp
2. Zamysł projektu
3. Zastosowane technologie
4. Obiekt
5. Sterowanie
6. Testowanie z wykorzystaniem programu typu sniffer
7. Wnioski
8. Referencje

1. Wstęp

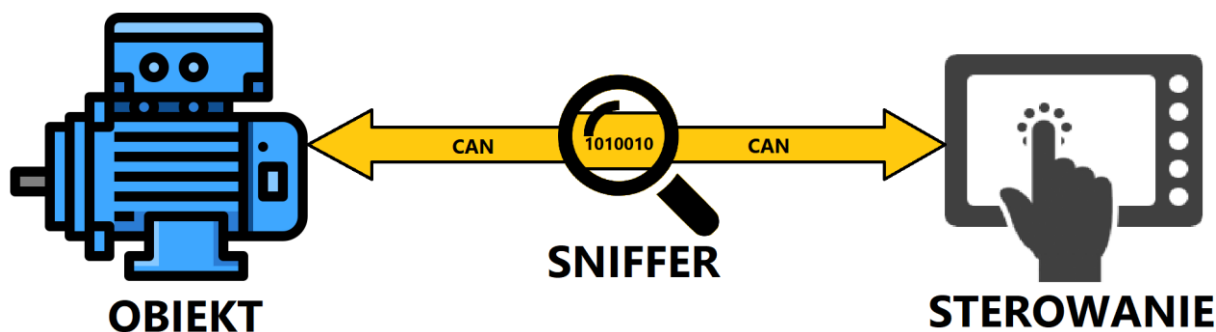
W ramach projektu „Projektowanie elektronicznych układów sterowania” napisane zostały programy reprezentujące idee sterowania silnikiem elektrycznym z wykorzystaniem interfejsu operatorskiego (HMI). Metoda komunikacji została oparta na szeregowej magistrali CAN.

2. Zamysł projektu

Zamysł projektu obejmuje umożliwienie użytkownikowi zadanie pożądanej prędkości oraz wizualizację danych dotyczących aktualnego stanu silnika elektrycznego. Symulacja odbywać się będzie na systemie Microsoft Windows, stąd możliwość obsługi wielowątkowości, a tym samym działanie dwóch programów jednocześnie.

Ponieważ aplikacje nie będą ze sobą powiązane w sposób bezpośredni, niezbędne jest zastosowanie narzędzia umożliwiającego zapisywanie i przesyłanie danych. W tym celu użyta zostanie wirtualna magistrala CAN symulująca pracę rzeczywistej aparatury.

Dodatkowo, dane z magistrali powinny zostać odczytywane przez oprogramowanie typu sniffer. Poniższy rysunek prezentuje zamysł projektu:



Rysunek 1 – Zamysł projektu

3. Zastosowane technologie

Napisanie aplikacji nie byłoby możliwe bez zastosowania wspieranych i popularnych technologii. Dlatego zdecydowaliśmy się na rozwiązania Microsoft .NET (aplikacje) oraz produkty firmy Kvaser (obsługa CAN) w tym:

1. Sterowanie – Aplikacja graficzna z zastosowaniem Windows Forms w wersji .NET Framework 4.6.2 dla języka C#,
2. Obiekt – Aplikacja konsolowa .NET Framework 4.6.2 dla języka C#,
3. Obsługa magistrali CAN – KVASER, gdzie:
 - Kvaser CAN hardware – sterowniki do obsługi magistrali,
 - Kvaser CANLib SDK – narzędzia, biblioteka dla języka C#,
 - Kvaser CanKing – oprogramowanie typu sniffer.

Dodatkowo, w celu urozmaicenia prezentacji danych na panelu sterowania użytkownika, wykorzystaliśmy komponent graficzny miernika firmy Syncfusion.

4. Obiekt

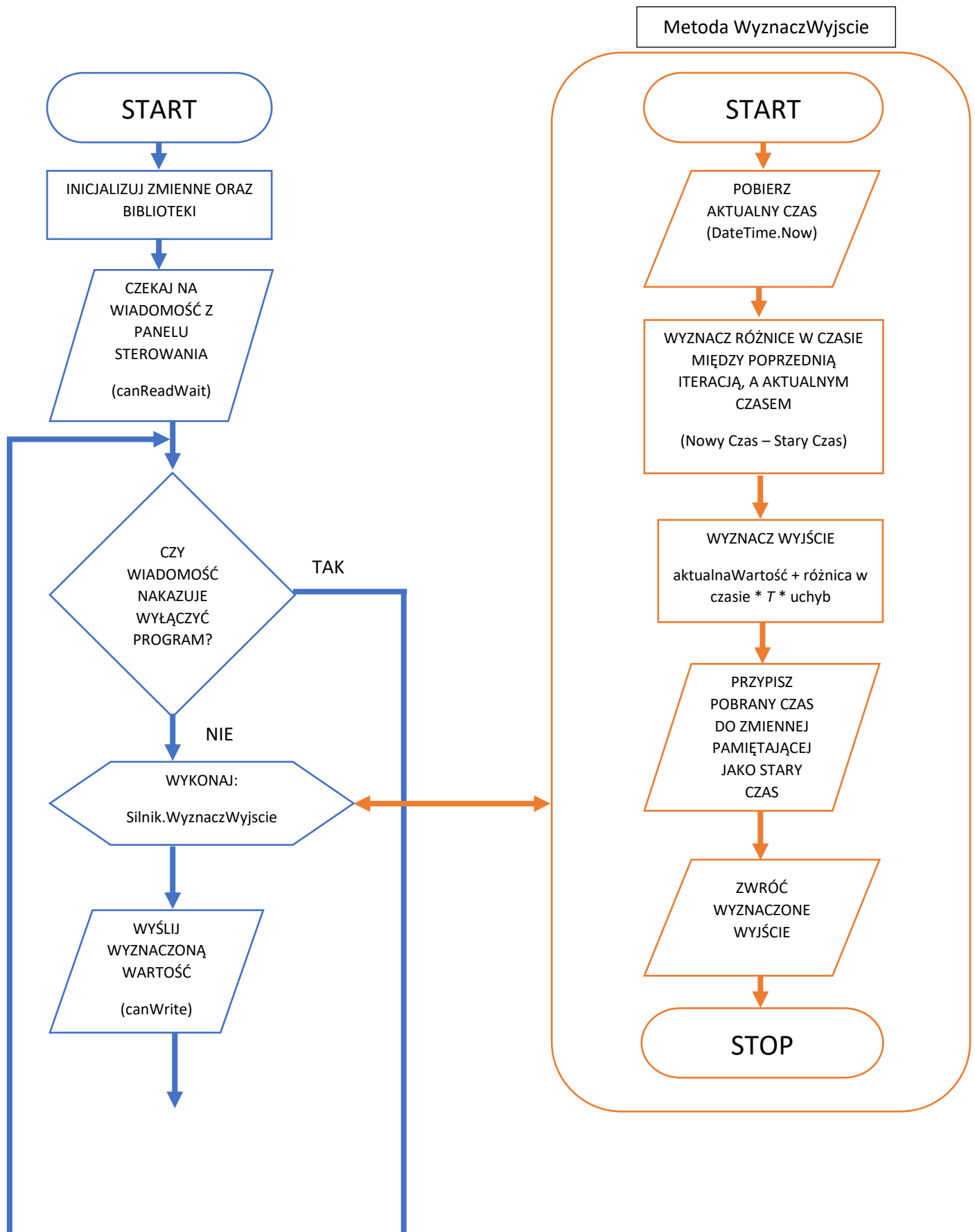
Aplikację „Obiekt” należy rozumieć jako obiekt inercyjny I rzędu o transmitancji danej wzorem:

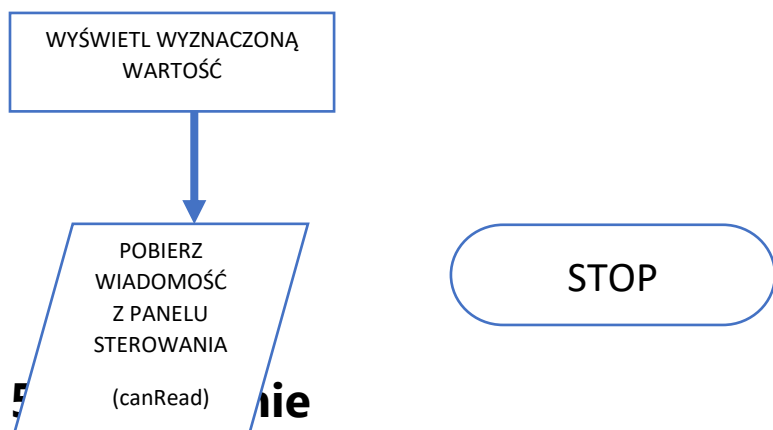
$$G(s) = \frac{1}{Ts + 1}$$

gdzie stała czasowa T może być regulowana w zależności od symulacyjnych potrzeb użytkownika.

Na program składają się 2 klasy, gdzie za punkt wejścia (metoda Main()) odpowiada wewnętrzna klasa **Program**. Klasa **Silnik** przechowuje właściwość stałej czasowej obiektu oraz metodę *WyznaczWyjscie(error)* zwracającą obliczoną prędkość w danej chwili czasowej. Implementacja ta wynika z zastosowania struktury DateTime biblioteki netstandard. Pozwala to zwiększenie dokładności zapisu zachowania obiektu.

Schemat blokowy na następnej stronie przedstawia algorytm działania aplikacji.



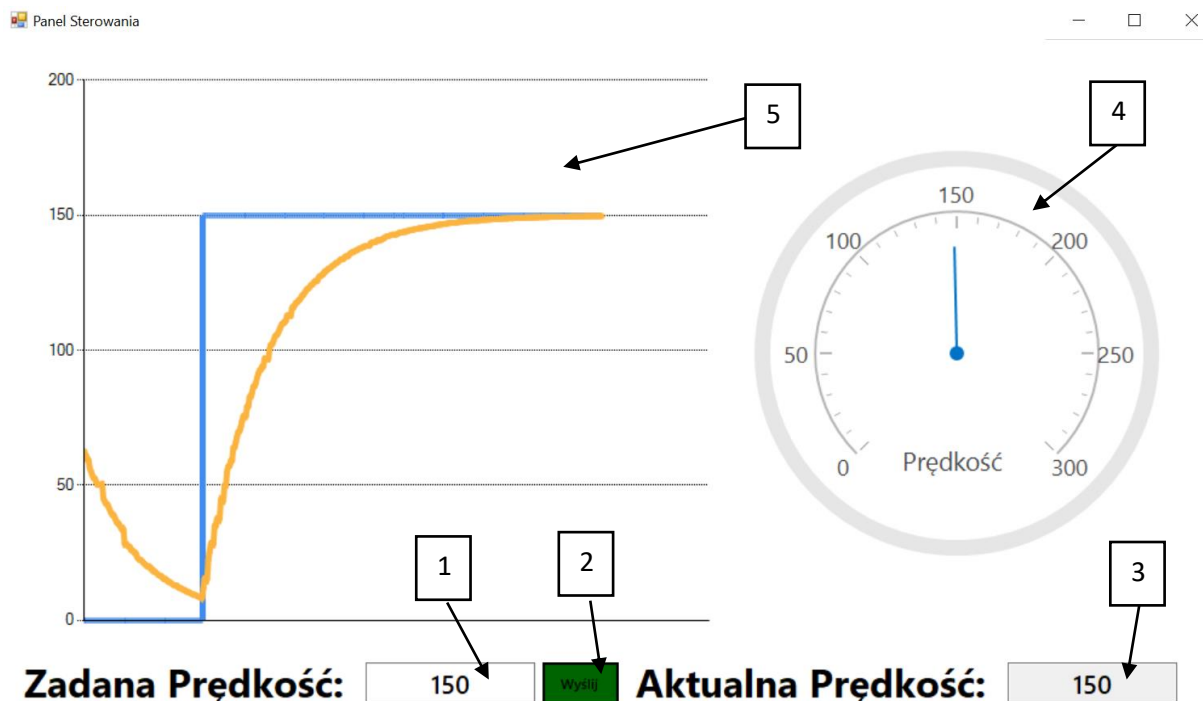


Aplikacja „Sterowanie” jest interfejsem użytkownika służącym do wyświetlania aktualnej prędkości obiektu oraz wysyłania nowych wartości zadanych.

Proces tworzenia wizualnej odsłony programu oparty jest na gotowych komponentach i customizacji z wykorzystaniem narzędzi dla .NET Windows Forms w Visual Studio 2022. Dlatego też część kodu nie będzie opisana, gdyż jego funkcjonalność nie jest związana bezpośrednio z zastosowaniem komunikacji CAN.

Faktyczny program zawarty jest w pliku *Program.cs* oraz *SterowanieForm.cs*, gdzie pierwszy z nich odpowiada za inicjalizację aplikacji oraz biblioteki CANLib. Logika odbywa się w drugiej klasie.

Interfejs panelu sterowania prezentuje się następująco:



Rysunek 2 – Interfejs panelu sterowania

W dalszej części omówione zostaną poszczególne fragmenty kodu zgodnie z numeracją przedstawioną na Rysunku 2. Jednakże w pierwszej kolejności należy przyjrzeć się znaczeniu pól globalnych:

```
10 namespace Sterowanie
11 {
12     3 references
13     public partial class SterowanieForm : Form
14     {
15         Thread aktualizujPolaDlaNowejPredkosci;
16         byte[] zadanaPredkosc = new byte[8];
17         byte[] aktualnaPredkosc = new byte[8];
18         ulong xSeries = 0;
19         int Channel0 = CanLib.canOpenChannel(0, CanLib.canOPEN_ACCEPT_VIRTUAL);
20     }
```

Rysunek 3 – Pola globalne klasy SterowanieForm

Dla naszej aplikacji tworzony jest wątek „*aktualizujPolaDlaNowejPredkosci*” który będzie odpowiadał za cykliczne pobieranie danych z obiektu. Dodatkowo, zainicjalizowany zostaje kanał „0” dla komunikacji CAN. Od tego momentu „*Channel0*” będzie otwarty i rozumiany jako „wirtualny”, bez bezpośredniego przełożenia na aparaturę fizycznie występującą. Dla niekonfigurowanego kanału, przyjmowana jest prędkość 500kbit/s przesyłania danych.

Konstruktor klasy, zawiera inicjalizację wszystkich komponentów aplikacji oraz zezwala na podłączenie kanału „0” do magistrali komunikacyjnej. Rozpoczyna się przesyłanie pierwszej wiadomości do obiektu. Zgodnie z schematem blokowym aplikacji „Obiekt”, program czeka na pierwszą wiadomość, w tym przypadku o treści „START”. Dodatkowo tworzona jest instancja wątku dla metody *AktualizujPolaDlaNowejPredkosci* oraz zostaje ona wywołana.

```
20 1 reference
21 public SterowanieForm()
22 {
23     InitializeComponent();
24     CanLib.canBusOn(Channel0);
25     CanLib.canWrite(Channel0, 10, Encoding.ASCII.GetBytes("START"), 8, 0);
26
27     Wykres.ChartAreas[0].AxisX.Minimum = 0;
28     Wykres.ChartAreas[0].AxisX.Maximum = 500;
29     Wykres.ChartAreas[0].AxisX2.Minimum = 0;
30     Wykres.ChartAreas[0].AxisX2.Maximum = 500;
31
32     aktualizujPolaDlaNowejPredkosci = new
33     Thread(AktualizujPolaDlaNowejPredkosci);
34     aktualizujPolaDlaNowejPredkosci.Start();
35 }
```

Rysunek 4 – Konstruktor klasy SterowanieForm

Jeżeli natomiast użytkownik wyłączy aplikację to zatrzymany zostanie proces pobierania i wyświetlania, program „Sterowanie” wyśle informacje do obiekt o zaprzestanie pracy, a kanał „0” odłączy się od magistrali.

```

105     1 reference
106     private void SterowanieForm_FormClosing(object sender, FormClosingEventArgs e)
107     {
108         aktualizujPolaDlaNowejPredkosci.Abort();
109         CanLib.canWrite(Channel0, 10, Encoding.ASCII.GetBytes("STOP"), 8, 0);
110         CanLib.canBusOff(Channel0);
111     }

```

Rysunek 5 – Metoda wywoływana przy zamknięciu aplikacji

Referencja do punktu 1 (Rysunek 2). Pole tekstowe *ZadanaPredkoscValue*

```

46     1 reference
47     private void ZadanaPredkoscValue_KeyPress(object sender, KeyPressEventArgs e)
48     {
49         if(System.Text.RegularExpressions.Regex.IsMatch(ZadanaPredkoscValue.Text, "[^0-9]"))
50         {
51             MessageBox.Show("Zadana Prędkość może być tylko liczbą naturalną!");
52             ZadanaPredkoscValue.Text = ZadanaPredkoscValue.Text.Remove(ZadanaPredkoscValue.Text.Length - 1);
53         }

```

Rysunek 6 – Metoda pola tekstowego *ZadanaPredkoscValue*

Pole tekstowe zostało zabezpieczone przed próbą zadania wartości prędkości innej niż liczba naturalna. W tym celu metoda dla wyrażeń regularnych sprawdza czy użytkownik nie wpisał znaków specjalnych bądź liter. Jeżeli warunek nie zostanie spełniony (wpisany tekst nie spełnia wymagań) to wyświetli się stosowny komunikat, a znak usunięty.

Referencja do punktu 2 (Rysunek 2). Przycisk *WyslijButton*

```

36     1 reference
37     private void WyslijButton_Click(object sender, EventArgs e)
38     {
39         bool isZadanaPredkosc = Double.TryParse(ZadanaPredkoscValue.Text, out double zadanaPredkoscDouble);
40         if (isZadanaPredkosc)
41         {
42             zadanaPredkosc = BitConverter.GetBytes(zadanaPredkoscDouble);
43             CanLib.canWrite(Channel0, 10, zadanaPredkosc, 8, 0);
44         }

```

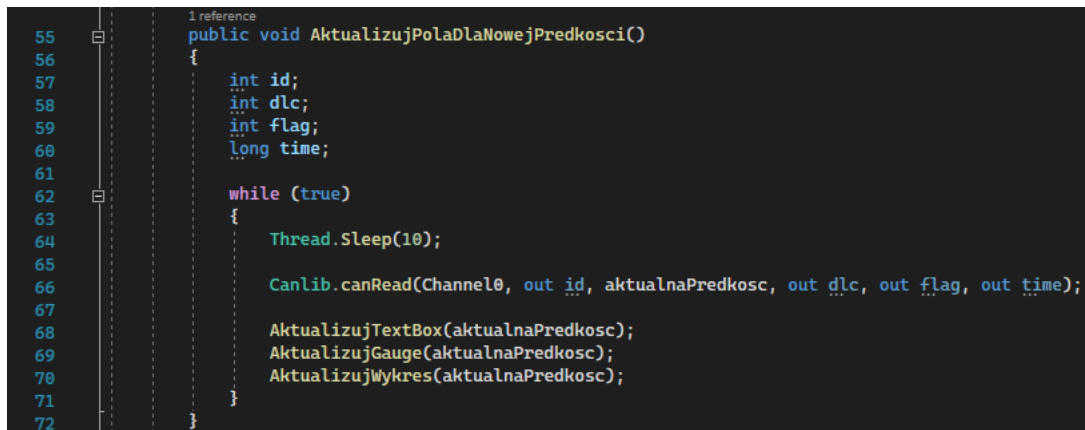
Rysunek 7 – Metoda przycisku *WyslijButton*

Przycisk pozwala na wywołanie akcji wysłania nowej wartości prędkości zadanej. W pierwszej kolejności sprawdzana jest struktura z pola tekstowego. Jeżeli możliwa jest konwersja z typu String na double to *zadanaPredkoscDouble* zapisywana jest w postaci 8 bajtów i wysyłana do obiektu poprzez kanał „0”. Metoda *canWrite(...)* przyjmuje kolejno:

- numer kanału,
- identyfikator urządzenia, z którego wysłana została wiadomość,
- wiadomość,
- długość wiadomości (dla naszego przypadku 8 bajtów),
- flaga (nieużywana, stąd domyślnie wartość 0).

Dlaczego dokonujemy konwersji na Double skoro Regex pola tekstowego zabezpiecza aplikację przed innymi wartościami niż liczby naturalne? Powodem jest długość wiadomości (ilość bajtów). Przyjmowany jest najdłuższy zakres 8 bajtów. Przy konwersji z wykorzystaniem *BitConverter* przeciążenie metody dla typu *Int32* buduje tylko 4 bajty. Operacje na danych bajtowych są mniej optymalne niż wykorzystanie zmiennej typu *Double*.

Referencja do metody *AktualizujPolaDlaNowejPredkosci*



```

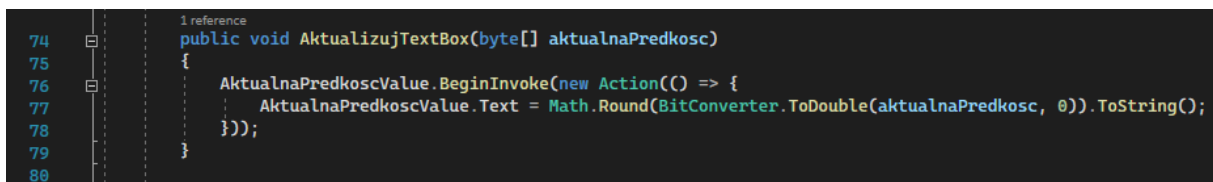
55 1 reference
56 public void AktualizujPolaDlaNowejPredkosci()
57 {
58     int id;
59     int dlc;
60     int flag;
61     long time;
62
63     while (true)
64     {
65         Thread.Sleep(10);
66
67         Canlib.canRead(Channel0, out id, aktualnaPredkosc, out dlc, out flag, out time);
68
69         AktualizujTextBox(aktualnaPredkosc);
70         AktualizujGauge(aktualnaPredkosc);
71         AktualizujWykres(aktualnaPredkosc);
72     }

```

Rysunek 8 – Metoda dla stworzonego wątku

Metoda *AktualizujPolaDlaNowePredkosci()* wywoływana jest w konstruktorze poprzez stworzony wątek. Jej funkcjonalność należy określić jako nieskończoną pętlę odczytu danych z obiektu (aktualnej prędkości) oraz uaktualnieniu panelu sterowania.

Referencja do punktu 3 (Rysunek 2). Pole tekstowe *AktualnaPredkoscValue*



```

74 1 reference
75 public void AktualizujTextBox(byte[] aktualnaPredkosc)
76 {
77     AktualnaPredkoscValue.BeginInvoke(new Action(() => {
78         AktualnaPredkoscValue.Text = Math.Round(BitConverter.ToDouble(aktualnaPredkosc, 0)).ToString();
79     }));
80 }

```

Rysunek 9 – Metoda pola tekstowego *AktualnaPredkoscValue*

Dla pola tekstowego niezbędna jest konwersja z tablicy typu *byte* na typ *String*. Wykonywane jest to poprzez predefiniowany delegat *Action()*.

Referencja do punktu 4 (Rysunek 2). Komponent *Gauge*

```
81 | 1 reference  
82 | public void AktualizujGauge(byte[] aktualnaPredkosc)  
83 | {  
84 |     Gauge.BeginInvoke(new Action() =>  
85 |     {  
86 |         Gauge.Value = Convert.ToSingle(BitConverter.ToDouble(aktualnaPredkosc, 0));  
87 |     }  
88 |     ));  
89 | }
```

Rysunek 10 – Metoda komponentu *Gauge*

Komponent *Gauge* jest gotowym rozwiązaniem udostępnionym w wersji DEMO przez firmę Syncfusion. Logika metody jest analogiczna do pola tekstowego *AktualnaPredkoscValue*. Różnica polega na przyjmowanym typie. Możliwe jest tylko podawanie wartości typu float.

Referencja do punktu 5 (Rysunek 2). Wykres *Wykres*

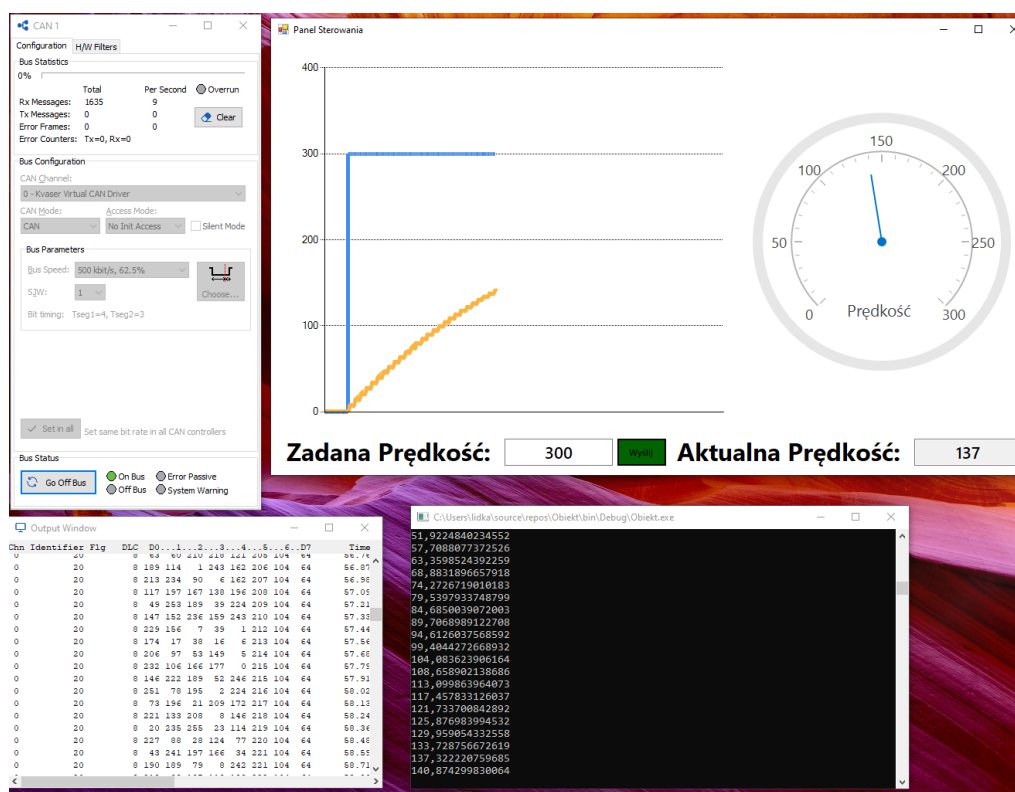
```
89 | 1 reference  
90 | public void AktualizujWykres(byte[] aktualnaPredkosc)  
91 | {  
92 |     xSeries += 1;  
93 |     Wykres.BeginInvoke(new Action() =>  
94 |     {  
95 |         if (xSeries > 500)  
96 |         {  
97 |             Wykres.Series["ZadanaPredkoscSeries"].Points.Clear();  
98 |             Wykres.Series["AktualnaPredkoscSeries"].Points.Clear();  
99 |             xSeries = 0;  
100 |         }  
101 |         Wykres.Series["ZadanaPredkoscSeries"].Points.AddXY(xSeries, BitConverter.ToDouble(zadanaPredkosc, 0));  
102 |         Wykres.Series["AktualnaPredkoscSeries"].Points.AddXY(xSeries, BitConverter.ToDouble(aktualnaPredkosc, 0));  
103 |     }  
104 |     ));  
105 | }
```

Rysunek 11 – Metoda wykresu *Wykres*

Animacja wykresu została rozwiązana poprzez cykliczne czyszczenie danych. Kolejne punkty wartości zadanej i aktualnej zapisywane są w tablicy wykresu, a po przekroczeniu 500 punktów usuwane. Pozwala to na zachowanie ciągłości wizualizacji.

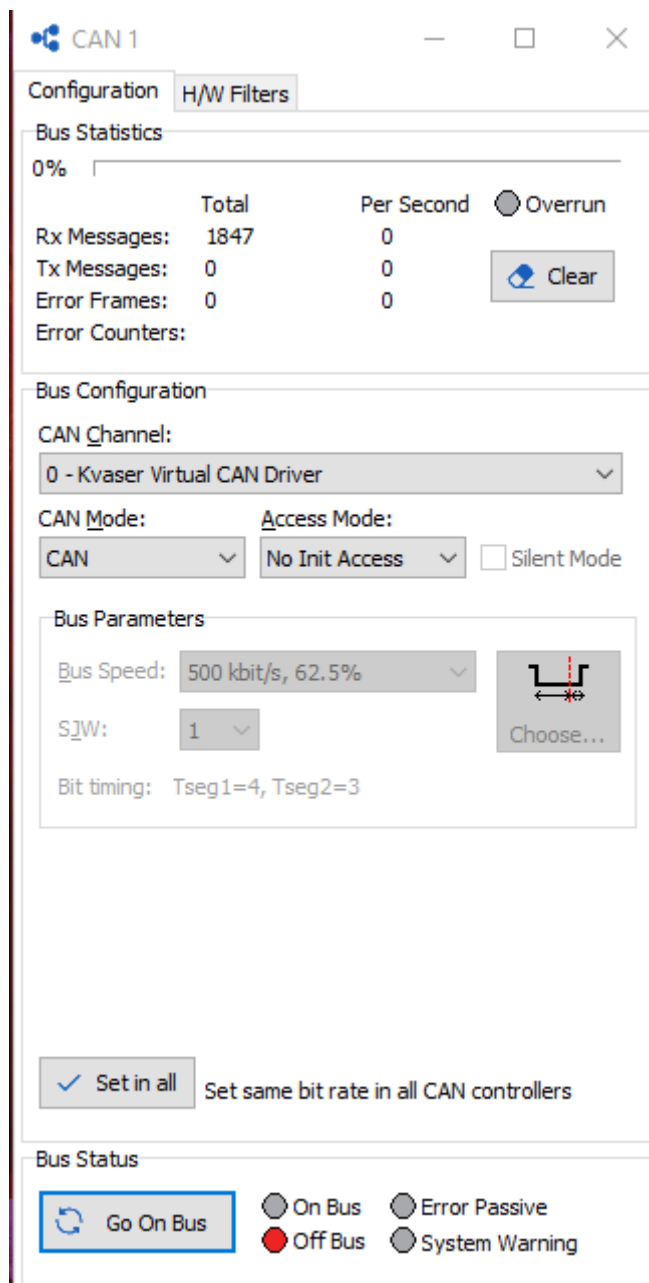
6. Testowanie z wykorzystaniem programu typu sniffer

Oprogramowanie Kvaser CanKing pozwala na podgląd wiadomości przesyłanych poprzez magistralę CAN. Dodatkowo możliwe jest sprawdzanie parametrów danej komunikacji. Z tego też powodu, sprawdziliśmy czy nasze aplikacje „Sterowanie” i „Obiekt” pracują poprawnie z zastosowaniem tego programu.



Rysunek 12 – Zrzut ekranu testów aplikacji z wykorzystaniem Kvaser CanKing

Na rysunku 12 przedstawione zostały okna aplikacji Kvaser CanKing (lewa strona) oraz „Obiekt” i „Sterowanie”. W trakcie badania kilku minutowego badania aplikacji przesłały 1847 wiadomości, stąd też możliwe jest bezproblemowe rozpatrzenie ich treści.



Rysunek 13 – Zrzut ekranu okna aplikacji Kvaser CanKing prezentująca magistralę komunikacyjną

Rysunek 13 pokazuje możliwość wyboru obserwowanego kanału (w tym przypadku użytkowany jest tylko kanał „0”) oraz parametry. Ponieważ wirtualny kanał nie był konfigurowany to standardowe ustawienia są widoczne, w tym prędkość 500kbit/s czy Bit timing.

Ze względu na charakter przesyłanych danych (wysyłanie nowych wiadomości przez Obiekt jest cykliczne, a w Panelu Sterowania zależne od zdarzeń) warto jest podjąć się analizy pierwszych oraz ostatnich rekordów przechwyconych podczas badania:

Output Window

Chn	Identifier	Flg	DLC	D0	1	2	3	4	5	6	D7	Time	Dir
0		10	8	83	84	65	82	84	0	0	0	15.371520	R
0		20	8	0	0	0	0	0	0	0	0	15.385450	R
0		20	8	200	120	127	27	2	0	0	0	15.496420	R
0		20	8	14	37	139	41	4	0	0	0	15.621570	R
0		20	8	245	67	207	234	5	0	0	0	15.730580	R
0		20	8	35	246	215	161	7	0	0	0	15.839950	R
0		20	8	158	156	167	78	9	0	0	0	15.949110	R

Rysunek 14 – Pierwsze 7 rekordów komunikacji po magistrali CAN

Na treść komunikatu danego rekordu składa się:

- Chn – channel, dany kanał. W naszym przypadku zawsze „0” gdyż używamy tylko jednego kanału
- Identifier – programistycznie ustalone Id. Przyjęty został zapis, że ID_{obiekt}=20, a ID_{sterowanie} = 10.
- Flg – flaga, w naszym przypadku zawsze pusty. Nie jest potrzebny przy tej transmisji.
- DLC – długość wiadomości liczona w bajtach. Przyjęty przez nas standard to 8 bajtów (pełen zakres).
- Od D0 do D7 – wartości na poszczególnych bajtach.

Pierwszy rekord opatrzony jest Id = 10, co oznacza, że komunikat został nadany przez „Sterowanie”. Wartości poszczególnych bajtów dekodowane z standardu ASCII wynoszą kolejno:

- D0 = (DEC)83 -> „S”,
- D1 = (DEC)84 -> „T”,
- D2 = (DEC)65 -> „A”,
- D3 = (DEC)82 -> „R”,
- D4 = (DEC)84 -> „T”,
- D5 = (DEC)0 -> „NULL”,
- D6 = (DEC)0 -> „NULL”,
- D7 = (DEC)0 -> „NULL”.

Tym samym prawdziwe jest, że komunikacja i praca obiektu powinna rozpocząć się od wiadomości „START” wysłanej przez Panel Sterowania.

Drugi rekord jest pierwszym komunikatem wysłanym przez Obiekt. Reprezentuje on wartość 0 czyli stan początkowy po uruchomieniu aplikacji.

Dekodowanie trzeciego rekordu należy rozpocząć od bajtu nr 7 (D7) dążąc do D0. Wartości wszystkich bajtów zostały zapisane w systemie dziesiętnym, a więc ich heksadecymalna postać ma wymiar:

- D7 = 0 -> 0x00,
- D6 = 0 -> 0x00,
- D5 = 0 -> 0x00,
- D4 = 2 -> 0x02,
- D3 = 27 -> 0x1B,
- D2 = 127 -> 0x7F,
- D1 = 120 -> 0x78,
- D0 = 200 -> 0xC8

Przesyłana wiadomość będzie więc miała wartość 0x000000021B7F78C8. Po konwersji na system dziesiętny jest to: **4.47192324596174580160724672498 * 10⁻³¹⁴**. Możemy więc mówić o „szumie” w układzie. Taki stan utrzyma się do momentu zadania konkretnej prędkości przez Panel Sterowania.

Analiza ostatnich rekordów transmisji powinna zakończyć się wyłączeniem programów:

0	20	8	97	1	205	213	58	170	114	64	226.210620 R
0	20	8	164	139	119	224	188	170	114	64	226.326990 R
0	20	8	176	148	132	138	65	171	114	64	226.448900 R
0	20	8	153	232	59	119	186	171	114	64	226.562970 R
0	10	8	83	84	79	80	0	0	0	0	226.595580 R

Rysunek 15 – Ostatnie 5 rekordów komunikacji po magistrali CAN

Wiadomość przedostatniego rekordu wysłanego przez Obiekt (ID = 20) wynosi: (HEX)0x4072ABBA773BE899 = (DEC)**298.733023866670521329069742933**. Wynika to z założenia, że ostatnią zadaną prędkością było 300, do której Obiekt dążył.

Ostatni komunikat został wysłany przez Panel Sterowania (ID = 10) i jest to wiadomość w kodzie ASCII, którego treść brzmi:

- D0 = (DEC)83 -> „S”,
- D1 = (DEC)84 -> „T”,
- D2 = (DEC)79 -> „O”,
- D3 = (DEC)80 -> „P”,
- D4 = (DEC)0 -> „NULL”,
- D5 = (DEC)0 -> „NULL”,
- D6 = (DEC)0 -> „NULL”,
- D7 = (DEC)0 -> „NULL”.

Ostatnia wiadomość została przesłana prawidłowo i zakończyła pracę aplikacji Obiekt.

7. Wnioski

Podsumowując niniejszy projekt, można z powodzeniem przyznać iż możliwe jest testowe tworzenie aplikacji mających na celu odwzorowanie fizycznej magistrali CAN. Programy typu sniffer umożliwiają sukcesywne przechwytywanie danych wysyłanych między układami.

Aplikacje takie jak tworzone na potrzeby projektu, mogą pomagać specjalistom w rozszyfrowywaniu i analizie układów rzeczywistych, maszyn czy rozwiązań związanych z mobilnością (przemysłem samochodowym), gdzie magistrala CAN jest uznawanym standardem.

Przykładem zastosowania aplikacji może być diagnostyka samochodowa przy wsparciu OBD (OBDII) czy ingerencja w komputer pokładowy na potrzeby np. zarządzania źródłem zasilania silnika (Benzyna/LPG).

8. Referencje

- Dokumentacja .NET Forms:
<https://docs.microsoft.com/pl-pl/dotnet/desktop/winforms/?view=netframeworkdesktop-4.8>
- Pakiety instalacyjne (sterowniki) Kvaser:
<https://www.kvaser.com/download/>
- Instalacja bibliotek Kvaser CANlib:
<https://www.kvaser.com/developer-blog/using-canlib-visual-studio-2019-c-net-standard-2-0/>
- Dokumentacja biblioteki Kvaser CANlib:
https://www.kvaser.com/canlib-webhelp/page_canlib.html
- Obsługa programu Kvaser CanKing:
<https://www.kvaser.com/canking/>
- Dokumentacja komponentu miernika Syncfusion:
<https://help.syncfusion.com/windowsforms/radial-gauge/radial-gauge>