WYKŁAD OTWIERAJĄCY — MODUŁ M6

Jeden algorytm, cztery języki

Ćwiczenie 43 — generator astabilny (migająca dioda): LAD, FBD, SCL i Arduino

Program działa. Ten wykład tłumaczy, DLACZEGO działa — i pokazuje, że dokładnie ta sama logika kryje się pod czterema zupełnie inaczej wyglądającymi zapisami.

Poziom	Klasa po M4/M5 — zna już C++/Arduino, logikę stykową, przekaźniki i timery czasowe (ELM07)
Czas	2 × 45 min (blok otwierający semestr) + praca własna nad pytaniami kontrolnymi
Cel	Zrozumieć MECHANIZM (nie tylko efekt) generatora astabilnego i przenieść go między 4 językami
Efekt	Uczeń tłumaczy tę samą logikę na LAD ↔ FBD ↔ SCL ↔ Arduino i przewiduje zachowanie po zmianie PT

Wstęp — dlaczego to jest pierwszy temat semestru

Na poprzednich zajęciach zbudowaliście działający układ: dioda mruga w rytmie „pół sekundy światła, pół sekundy przerwy". Program działa — widać to gołym okiem. Ale „działa" to dopiero połowa sukcesu zawodowego technika robotyka. Prawdziwe pytanie egzaminacyjne i prawdziwe pytanie na hali produkcyjnej brzmi: DLACZEGO to działa? Bo tylko ten, kto rozumie mechanizm, potrafi program naprawić, zmodyfikować pod nowe warunki albo przenieść na zupełnie inne urządzenie.

Ten wykład ma jeden cel: rozłożyć wasz program na czynniki pierwsze i pokazać, że dokładnie ten sam pomysł da się zapisać na cztery różne sposoby — jako drabinkę (LAD), jako klocki funkcyjne (FBD), jako tekst (SCL) oraz jako program mikrokontrolera (Arduino — blokowo i w C). To nie są cztery różne programy. To jest JEDEN program w czterech przebraniach. Kto to raz zrozumie, przez resztę roku nie będzie się bał żadnego nowego języka — bo pod spodem zawsze znajdzie to samo.

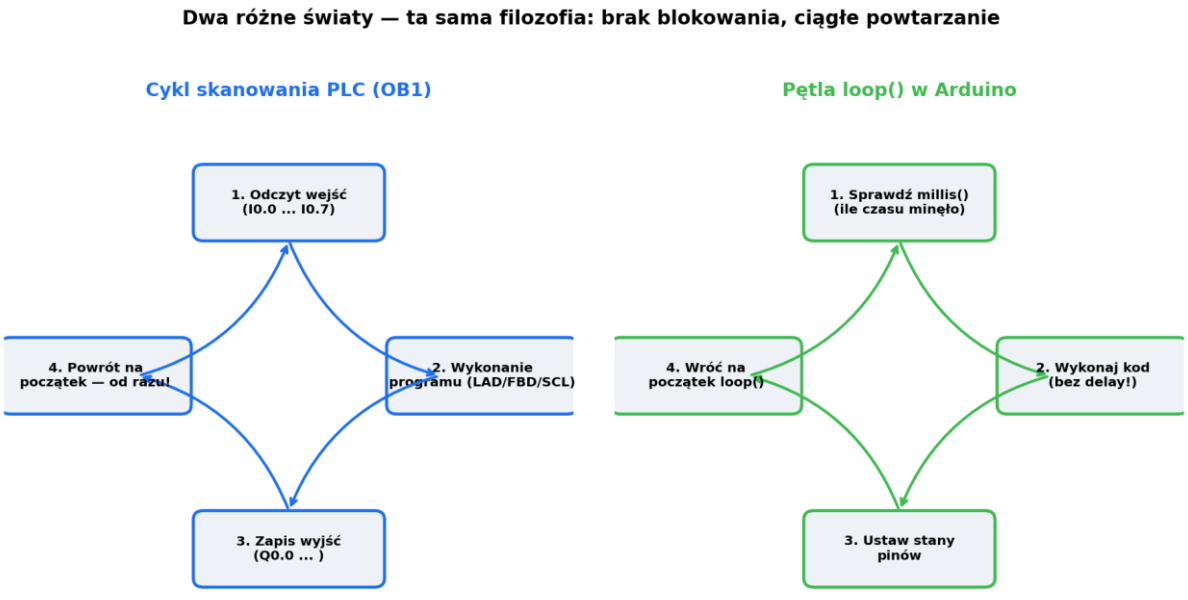
Materiał wyjściowy: ćwiczenie 43

Zadanie: zbudować generator astabilny (migającą diodę) na sterowniku S7-1200, bez korzystania z ustawień sprzętowych CPU — wyłącznie logiką programu, na dwóch timerach typu TON (Time On-Delay), tak aby rozwiązanie działało identycznie na każdym sterowniku z rodziny S7-1200/1500.

1. Wielka zasada: nic nie wolno zatrzymać

Zanim spojrzymy na sam program, musimy zrozumieć coś znacznie ważniejszego niż składnia LAD czy FBD — musimy zrozumieć, JAK w ogóle pracuje sterownik PLC. To jest fundament, bez którego cała reszta wykładu nie będzie miała sensu.

Sterownik PLC nie wykonuje programu „raz" i nie czeka bezczynnie. On bez przerwy powtarza jeden i ten sam cykl: odczytuje wszystkie wejścia, wykonuje CAŁY program od góry do dołu, zapisuje wszystkie wyjścia — i natychmiast wraca na początek. Taki cykl (tzw. skan) trwa zwykle kilka do kilkudziesięciu milisekund i powtarza się bez końca, tysiące razy na minutę. Arduino, wykonując funkcję loop(), robi dokładnie to samo — różnica jest tylko w nazewnictwie.



Rys. 1 — cykl skanowania PLC i pętla loop() Arduino to ta sama filozofia działania

Dlaczego nie można po prostu użyć „czekaj 500 ms”?

Gdyby jedna instrukcja w programie PLC potrafiła zatrzymać cały cykl na 500 ms, to przez te 500 ms sterownik NIE odczytałby żadnego innego wejścia — łącznie z przyciskiem awaryjnego STOP! Cała maszyna „zamarłaby” na czas trwania tej jednej instrukcji. Dlatego w językach sterowników PLC nie istnieje odpowiednik „delay()”. Zamiast tego używa się timerów, które odmierzają czas W TLE, nie zatrzymując reszty programu ani na chwilę.

To jest właśnie powód, dla którego w ćwiczeniu 43 potrzebujemy AŻ DWÓCH timerów zamiast jednego „czekaj”. Każdy z nich w danej chwili tylko „patrzy”, ile czasu minęło — a program leci dalej, cykl po cyklu, bez zatrzymania.

2. Analiza uniwersalna — trzy równania rządzące wszystkim

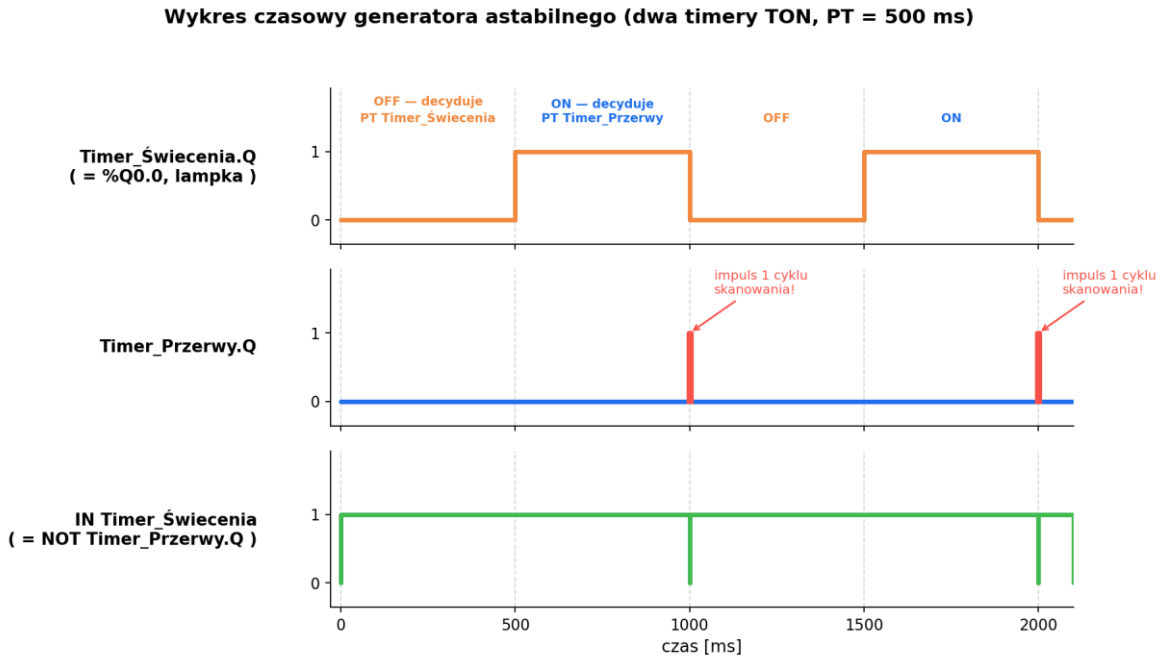
Zanim przejdziemy do konkretnego języka, zapiszmy działanie układu w sposób niezależny od jakiegokolwiek języka programowania — za pomocą trzech równań logicznych. To jest właśnie ten „szkielet”, który w dalszej części wykładu ubierzemy kolejno w LAD, FBD, SCL i Arduino.

Przypomnienie: blok TON (Time On-Delay) ma wejście IN i wyjście Q. Dopóki IN = 1, wewnętrzny licznik czasu ET rośnie. Gdy ET osiągnie wartość PT, wyjście Q ustawia się na 1 i tak zostaje, dopóki IN = 1. W chwili, gdy IN spadnie na 0 — Q natychmiast wraca do 0, a ET zeruje się. TON nie ma pamięci: całkowicie „zapomina”, ile zdążył doliczyć.

Trzy równania układu

```
IN(Timer_Świecenia)  =  NOT Timer_Przerwy.Q
IN(Timer_Przerwy)    =  Timer_Świecenia.Q
%Q0.0  (lampka)      =  Timer_Świecenia.Q
```

Zobaczcie: to są DOKŁADNIE te same trzy linijki, które za chwilę zobaczycie jako styki w LAD, jako bloki w FBD i jako kod w SCL. Zmieni się tylko notacja — treść ani razu.



Rys. 2 — wykres czasowy: dlaczego Timer_Przerwy.Q to w rzeczywistości bardzo krótki impuls, a nie stan trwający pół sekundy

Krok po kroku — co się dzieje w kolejnych chwilach

1. t = 0 ms (start): oba timery mają Q = 0. NOT Timer_Przerwy.Q = NOT 0 = 1, więc IN Timer_Świecenia = 1 — Timer_Świecenia zaczyna liczyć od zera. Lampka (%Q0.0 = Timer_Świecenia.Q) jest zgaszona.
2. t = 0–500 ms: Timer_Świecenia liczy (ET rośnie 0→500 ms), jego Q wciąż = 0, więc IN Timer_Przerwy = 0 — Timer_Przerwy stoi w miejscu. Lampka zgaszona przez cały ten czas.
3. t = 500 ms: Timer_Świecenia osiąga PT = 500 ms → jego Q przeskakuje na 1. Lampka zapala się. Jednocześnie ten sam sygnał (Timer_Świecenia.Q = 1) staje się nowym IN dla Timer_Przerwy — ten zaczyna liczyć od zera.
4. t = 500–1000 ms: Timer_Przerwy liczy (ET rośnie 0→500 ms). Jego Q wciąż = 0, więc styk NC w Network 1 pozostaje zamknięty, IN Timer_Świecenia dalej = 1 — lampka pozostaje zapalona przez cały ten czas.
5. t = 1000 ms: Timer_Przerwy osiąga PT = 500 ms → jego Q przeskakuje na 1. To natychmiast otwiera styk NC w Network 1 (bo NOT 1 = 0) — IN Timer_Świecenia spada do 0. TON „zapomina” swój postęp: ET i Q Timer_Świecenia wracają do 0 w tej samej chwili. Lampka gaśnie.
6. Skutek uboczny: skoro Timer_Świecenia.Q właśnie spadło do 0, to IN Timer_Przerwy też spada do 0 — więc Timer_Przerwy RÓWNIEŻ natychmiast się resetuje (jego Q wraca do 0). Dlatego na wykresie czasowym Timer_Przerwy.Q widać nie „pół sekundy jedynki”, tylko krótkiziutki impuls — trwający dokładnie tyle, ile jeden cykl skanowania sterownika.
7. t = 1000 ms: układ jest w dokładnie takim samym stanie, jak przy t = 0. Cały cykl zaczyna się od nowa — i tak w kółko, bez końca.

Pułapka nazewnictwa — najważniejszy wniosek z tej analizy!

Nazwy sugerują, że „Timer_Świecenia” odmierza czas świecenia, a „Timer_Przerwy” — czas przerwy. To pułapka! Ze szczegółowej analizy powyżej wynika coś odwrotnego:

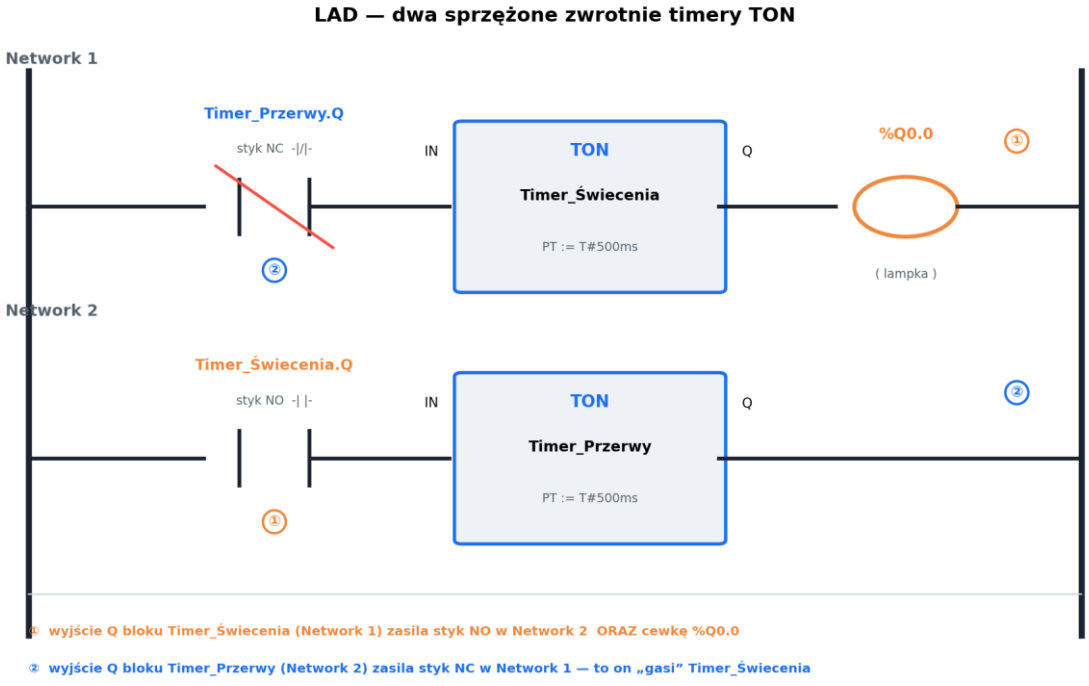
• Czas, przez jaki lampka jest ZGASZONA, zależy od PT ustawionego w Timer_Świecenia (bo to on liczy, zanim cokolwiek się zapali).

• Czas, przez jaki lampka ŚWIECI, zależy od PT ustawionego w Timer_Przerwy (bo to on liczy, gdy lampka już się pali, i to jego przepełnienie ją gasi).

Sprawdźcie sami: gdybyśmy zmienili PT Timer_Przerwy z 500 ms na 200 ms (zostawiając Timer_Świecenia bez zmian), lampka nadal zapalałaby się po 500 ms ciemności — ale świeciłaby już tylko 200 ms. To świetne pytanie na sprawdzian.

3. Język 1 — LAD (Ladder Diagram, „drabinka”)

LAD to najbardziej „elektryczny” z języków sterowników — rysuje się go tak, jakby to był schemat obwodu ze stykami i cewkami, w którym „prąd” płynie od lewej szyny zasilającej do prawej. Styk zwierny (NO, -| |-) przepuszcza sygnał, gdy powiązany z nim bit = 1. Styk rozwierny (NC, -|/|-) przepuszcza sygnał wtedy, gdy bit = 0 — działa więc jak logiczne NOT.



Rys. 3 — pełny program w LAD: dwie sieci (network) w pełni realizujące trzy równania z rozdziału 2

Jak to zbudować w TIA Portal — krok po kroku

1. Wstaw pusty blok (Shift+F5), wpisz TON, nadaj mu nazwę Timer_Świecenia — to Network 1.
2. Na wejściu IN tego bloku wstaw styk normalnie zamknięty -|/|- (NC) i przypisz mu adres Timer_Przerwy.Q.
3. Poniżej wstaw drugi blok TON, nazwij go Timer_Przerwy — to Network 2.
4. Na wejściu IN drugiego bloku wstaw styk normalnie otwarty -| |- (NO) i przypisz mu adres Timer_Świecenia.Q.
5. Do wyjścia Q pierwszego bloku (Timer_Świecenia), w tej samej sieci (Network 1), dołącz cewkę wyjścia fizycznego %Q0.0.
6. W obu blokach TON wpisz czas PT = T#500ms.
7. Wgraj program, przełącz się w tryb podglądu (okulary) i sprawdź, czy wartości ET rzeczywiście „biegają” w rytm 500 ms.

> **Uwaga sprzętowa — sterownik 1215 DC/DC/DC**
>
> Jeśli program w podglądzie wygląda poprawnie (ET rośnie, Q przełącza się), a fizyczna dioda mimo to nie miga — sprawdźcie zasilanie sekcji wyjść! W sterownikach z wyjściami tranzystorowymi (…/DC) na listwie zaciskowej trzeba doprowadzić osobne 24 V do zacisków 3L+ / 3M (lub 1L+ / 1M). Bez tego tranzystory

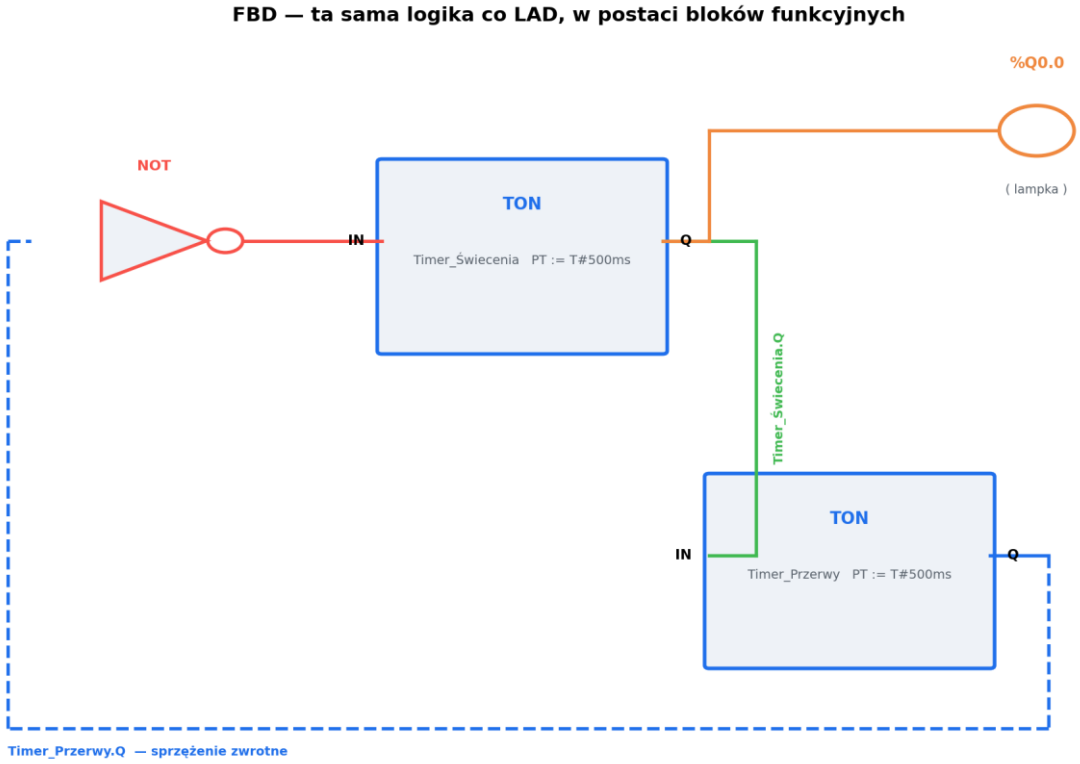
> wyjściowe nie mają z czego „wystawić” sygnału na zewnątrz, mimo że logika w środku sterownika działa bez zarzutu.

Tłumaczenie na znany już język — logikę stykową

To wcale nie jest dla was nowość! W module poprzednim (ELM07) budowaliście już blokady wzajemne i układy z podtrzymaniem na stycznikach i przekaźnikach — styk NC „gaszący” obwód i styk NO „podtrzymujący” go to dokładnie ten sam pomysł, tylko teraz zamiast fizycznych przewodów rysujecie go na ekranie.

4. Język 2 — FBD (Function Block Diagram)

FBD to ta sama logika, tylko zamiast „prądu płynącego przez styki” mamy sygnały płynące przewodami między blokami funkcyjnymi. Styk NC z LAD zamienia się w bramkę NOT, styk NO — po prostu w zwykły przewód (bo NO nic nie neguje, tylko przekazuje sygnał dalej).



Rys. 4 — ten sam program w FBD: bramka NOT + dwa bloki TON + cewka wyjścia

Słownik LAD → FBD

Element w LAD	Odpowiednik w FBD	Znaczenie logiczne
styk NO -| |-	zwykły przewód (brak bramki)	przekazanie wartości bez zmian
styk NC -|/|-	bramka NOT	zaprzeczenie (negacja) wartości
szeregowe połączenie styków	bramka AND	iloczyn logiczny
równoległe połączenie styków	bramka OR	suma logiczna
cewka ( )	przypisanie do zmiennej wyjściowej	zapisanie wyniku do bitu/adresu

W TIA Portal wiele sieci można przełączyć między widokiem LAD i FBD jednym kliknięciem — bo to naprawdę ten sam program, tylko inaczej narysowany. To najlepszy dowód na główną tezę tego wykładu.

5. Język 3 — SCL (Structured Control Language)

SCL to język tekstowy — pierwszy krok w stronę tego, co już trochę znacie z Arduino/C++. Tu nie rysujemy już styków ani bloków: piszemy nasze trzy równania niemal dosłownie, jako wywołania bloków funkcyjnych i przypisania.

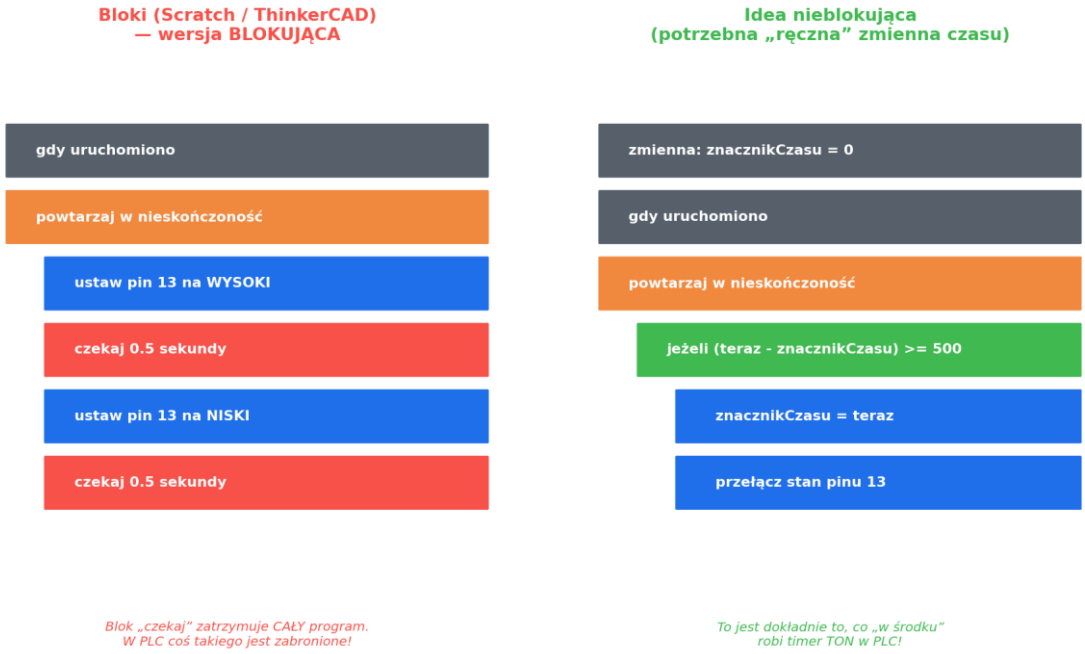
```
// Wywołanie instancji TON "Timer_Świecenia":
// IN otrzymuje wprost wartość logiczną NOT Timer_Przerwy.Q
"Timer_Świecenia"(IN := NOT "Timer_Przerwy".Q,
                  PT := T#500ms);

// Wywołanie instancji TON "Timer_Przerwy":
// IN otrzymuje wprost wartość Timer_Świecenia.Q
"Timer_Przerwy"(IN := "Timer_Świecenia".Q,
                PT := T#500ms);

// Przypisanie do wyjścia fizycznego
"Q0.0" := "Timer_Świecenia".Q;
```

Porównajcie to z równaniami z rozdziału 2 — to jest ten sam zapis, tylko z dodanym średnikiem i słowem kluczowym IN :=. Żadnych nowych pomysłów, tylko nowa „czcionka”.

6. Język 4 — Arduino / ThinkerCAD

Teraz przenosimy dokładnie ten sam problem na mikrokontroler. Celowo pokażemy TRZY kolejne wersje — od najbardziej intuicyjnej (i błędnej z punktu widzenia „filozofii PLC”) do takiej, która jest wiernym odpowiednikiem naszych dwóch timerów.

6.1. Wersja blokowa (Scratch / ThinkerCAD) — intuicyjna, ale blokująca



Rys. 5 — po lewej: naturalne podejście blokowe z „czekaj”. Po prawej: idea, której PLC używa zawsze.

Blok „czekaj X sekund” w ThinkerCAD (i instrukcja delay() w C++) zatrzymuje wykonywanie całego programu na ten czas — dokładnie to, czego w rozdziale 1 zabroniliśmy sterownikowi PLC robić! Dla prostego mrugania

jedną diodą to zwykle nie szkodzi. Ale gdyby w tym samym programie Arduino miało jednocześnie obsługiwać np. przycisk bezpieczeństwa — przez te 500 ms delay() ono by go nie zauważyło. To ta sama pułapka, co brak „czekaj” w PLC.

6.2. Kod C — wersja blokująca (dla porównania)

```
void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // zapal diodę
  delay(500);               // ZATRZYMAJ wszystko na 500 ms
  digitalWrite(13, LOW);    // zgaś diodę
  delay(500);               // ZATRZYMAJ wszystko na kolejne 500 ms
}
```

6.3. Kod C — wersja nieblokująca (millis) — prawdziwy odpowiednik dwóch timerów PLC

```
unsigned long znacznikCzasu = 0;   // odpowiednik ET – "od kiedy liczymy"
const long interwal = 500;         // odpowiednik PT – czas do odczekania
bool stanDiody = LOW;

void setup() {
  pinMode(13, OUTPUT);
}

void loop() {
  unsigned long teraz = millis();  // zegar systemowy, nikt go nie zatrzymuje

  if (teraz - znacznikCzasu >= interwal) {   // czy minęło już PT?
    znacznikCzasu = teraz;                    // nowy "start" – reset timera
    stanDiody = !stanDiody;                   // przełącz stan (odpowiednik Q)
    digitalWrite(13, stanDiody);
  }
  // ... tutaj mogłby być kod obsługujący przycisk STOP –
  // i zawsze zdążyłby się wykonać w tym samym przebiegu loop()!
}
```

To jest kluczowy moment całego wykładu: wyrażenie teraz - znacznikCzasu to jest dosłownie ET timera TON. Warunek >= interwal to dosłownie porównanie ET z PT wewnątrz bloku TON. Funkcja loop() to dosłownie cykl skanowania sterownika. Arduino w tej wersji „udaje” PLC — i robi to bardzo wiernie.

6.4. Wersja rozszerzona — ta sama „pułapka nazewnictwa”, co w LAD

Żeby program w Arduino był NAPRAWDĘ wiernym tłumaczeniem naszych dwóch osobnych timerów (a nie tylko szczęśliwym przypadkiem symetrii 500/500), rozdzielmy czas świecenia i czas przerwy na dwie osobne stałe:

```
const unsigned long CZAS_OFF = 500;   // odpowiednik PT Timer_Świecenia
const unsigned long CZAS_ON  = 500;   // odpowiednik PT Timer_Przerwy

unsigned long znacznikCzasu = 0;
bool lampkaSwieci = false;

void setup() {
```

```
  pinMode(13, OUTPUT);
}

void loop() {
  unsigned long teraz = millis();
  unsigned long czasOczekiwania = lampkaSwieci ? CZAS_ON : CZAS_OFF;

  if (teraz - znacznikCzasu >= czasOczekiwania) {
    znacznikCzasu = teraz;
    lampkaSwieci = !lampkaSwieci;
    digitalWrite(13, lampkaSwieci ? HIGH : LOW);
  }
}
```

Teraz, dokładnie jak w LAD: zmiana wartości CZAS_OFF wpłynie na to, jak długo dioda jest zgaszona, a zmiana CZAS_ON — jak długo świeci. Ta sama zasada, ten sam „rozjazd" między intuicyjną nazwą a realnym działaniem — tylko tym razem w C, nie w drabince.

7. Tabela zbiorcza — jeden algorytm, cztery stroje

Element logiki	LAD	FBD	SCL	Arduino (C)
Zasilenie Timer_Świecenia	styk NC Timer_Przerwy.Q	bramka NOT	IN := NOT ...Q	if (!lampkaSwieci) ...
Zasilenie Timer_Przerwy	styk NO Timer_Świecenia.Q	zwykły przewód	IN := ...Q	if (lampkaSwieci) ...
Pomiar czasu	blok TON, ET/PT	blok TON, ET/PT	wywołanie TON(...)	millis() - znacznikCzasu
Wyjście fizyczne	cewka %Q0.0	cewka na końcu przewodu	"Q0.0" := ...Q	digitalWrite(13, ...)
Zatrzymanie programu?	nigdy (zabronione)	nigdy (zabronione)	nigdy (zabronione)	TYLKO przy delay() — unikać!

Najważniejsza myśl na wynos

Zapamiętajcie na cały rok

Nowy język programowania to nie jest nowa wiedza — to nowa gramatyka dla wiedzy, którą już macie. Jeśli rozumiecie, co ma się wydarzyć (te trzy równania), to LAD, FBD, SCL i C to już tylko kwestia słownictwa. Tej zasady będziemy używać przez cały rok — także wtedy, gdy przejdziemy do programowania robota w MELFA BASIC.

8. Pytania kontrolne (baza do sprawdzianu nr 1)

1. Dlaczego w programie sterownika PLC nie stosuje się instrukcji typu „czekaj”, tak jak delay() w Arduino?
2. Co się stanie z sygnałem %Q0.0, jeśli w Network 1 zamienimy styk NC na styk NO (bez żadnych innych zmian)?
3. Który z dwóch timerów naprawdę decyduje o czasie, przez jaki lampka jest ZGASZONA? Uzasadnij na podstawie analizy z rozdziału 2.
4. Ustawiono PT Timer_Świecenia = 300 ms, a PT Timer_Przerwy = 700 ms. Jak długo lampka będzie zgaszona, a jak długo zapalona w jednym pełnym cyklu?
5. Narysuj (lub opisz słownie) przebieg sygnału Timer_Przerwy.Q — dlaczego jest to krótki impuls, a nie stan trwający pół sekundy?
6. Przetłumacz równanie IN(Timer_Świecenia) = NOT Timer_Przerwy.Q na: a) styk w LAD, b) bramkę w FBD, c) linijkę kodu w SCL.
7. W kodzie Arduino z millis(): co dokładnie odpowiada zmiennej ET timera TON? A co odpowiada PT?
8. Dlaczego wersja Arduino z delay() nie nadaje się do programu, który równolegle musi obsługiwać przycisk bezpieczeństwa?
9. Sterownik 1215 DC/DC/DC ma poprawny program (widać to w podglądzie), ale dioda nie miga fizycznie. Co sprawdzamy w pierwszej kolejności?
10. Zaproponuj, jak za pomocą tych samych dwóch timerów (TON) zbudować sygnalizację z asymetrycznym miganiem: 100 ms światła, 900 ms przerwy.

Klucz odpowiedzi dla nauczyciela (skrót)

1) Zatrzymałoby to cały cykl skanowania i wszystkie inne operacje, w tym odczyt wejść bezpieczeństwa. 2) Układ przestałby oscylować — Timer_Świecenia zostałby zasilony na stałe (jeśli Timer_Przerwy.Q=0) lub nigdy (jeśli Timer_Przerwy.Q=1), zależnie od stanu początkowego — traci się sprzężenie zwrotne gaszące. 3) Timer_Świecenia — bo to on liczy ET, zanim Q0.0 w ogóle się załączy. 4) Zgaszona 300 ms, zapalona 700 ms. 5) Bo Timer_Przerwy resetuje się w tej samej chwili, w której jego IN (=Timer_Świecenia.Q) spada do 0 — a dzieje się to natychmiast po tym, jak Timer_Przerwy zdążyło przez moment osiągnąć Q=1. 7) ET = teraz - znacznikCzasu, PT = interwal/CZAS_ON/CZAS_OFF. 9) Zasilanie sekcji wyjść: zaciski 3L+/3M. 10) PT Timer_Świecenia = 900 ms (czas OFF), PT Timer_Przerwy = 100 ms (czas ON).