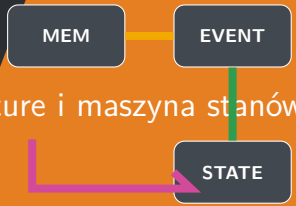
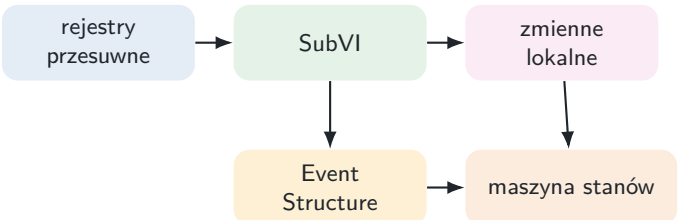
Program, który pamięta i reaguje

Rejestry przesuwne, SubVI, zmienne lokalne, Event Structure i maszyna stanów



Wprowadzenie do LabVIEW i układów kontrolno-pomiarowych ▪ Wykład 3
dr inż. Angelika Tefelska

Mapa dzisiejszego wykładu



- Cel: zbudować program, który **pamięta poprzedni stan**, reaguje na działania użytkownika i może wykonywać różne etapy algorytmu.
- To jest bezpośrednie przygotowanie do laboratoriów oraz miniprojektu z maszyny stanów.

1. Rejestry przesuwne

Jak przenieść informację z jednej iteracji do następnej?



Dlaczego sama pętla nie wystarcza?

Bez pamięci

W każdej iteracji obliczenia korzystają tylko z danych dostępnych **teraz**. Po zakończeniu iteracji lokalny wynik nie jest automatycznie wejściem kolejnej.

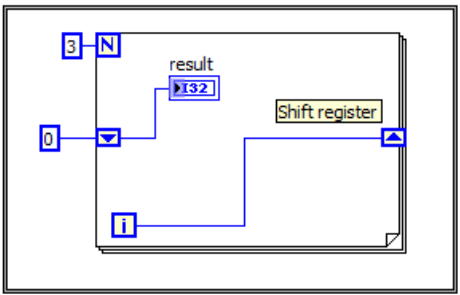
Z pamięcią

Często potrzebujemy wartości z poprzedniej iteracji: sumy, poprzedniego pomiaru, licznika, stanu programu lub kilku wcześniejszych próbek.



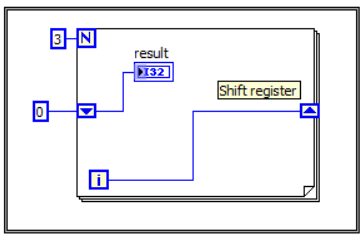
Rejestr przesuwny – idea

- Rejestr przesuwny działa na granicy pętli For lub While.
- **Lewy terminal** dostarcza wartość zapisaną w poprzedniej iteracji.
- **Prawy terminal** przyjmuje wartość, która ma zostać przekazana do następnej iteracji.
- Tworzymy go przez kliknięcie granicy pętli i wybranie *Add Shift Register*.



Rejestr z wartością początkową

- Do lewego terminala **z zewnątrz pętli** podłączamy wartość startową.
- W pierwszej iteracji lewy terminal zwraca właśnie tę wartość.
- Po każdej iteracji wartość z prawego terminala staje się „poprzednią" wartością dla kolejnej iteracji.



i	lewy terminal	prawy terminal
0	0	0
1	0	1
2	1	2

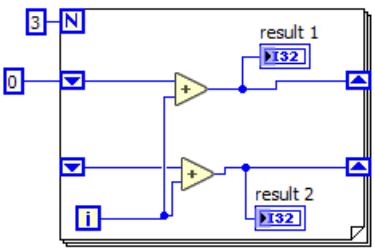
Wartości domyślne

Jeśli nie zadeklarowaliśmy wartości początkowej rejestru przesuwnego to zostanie użyta wartość domyślna dla danego typu:

- Numeric - 0
- Boolean - FALSE
- String - Pusty znak

Rejestr bez jawnej inicjalizacji

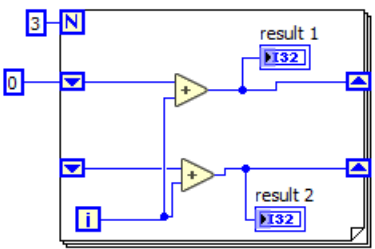
- Jeśli nie podłączymy wartości początkowej, rejestr staje się **niezainicjalizowany**.
- W takim wypadku przy pierwszym uruchomieniu, wartość z lewego rejestru jest równa wartości domyślnej dla danego typu.



Pierwsze uruchomienie programu						
i	lewy terminal	*result 1*	prawy terminal	lewy terminal	*result 2*	prawy terminal
0	0	0 (0+0)	0	0	0 (0+0)	0
1	0	1 (0+1)	1	0	1 (0+1)	1
2	1	3 (1+2)	3	1	3 (1+2)	3

Rejestr bez jawnej inicjalizacji

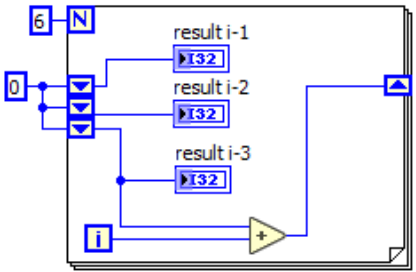
- W przypadku drugiego uruchomienia programu, wartość z lewego rejestru jest równa wartości z prawego rejestru z poprzedniego uruchomienia programu.



Pierwsze uruchomienie programu						
i	lewy terminal	*result 1*	prawy terminal	lewy terminal	*result 2*	prawy terminal
0	0	0 (0+0)	0	0	0 (0+0)	0
1	0	1 (0+1)	1	0	1 (0+1)	1
2	1	3 (1+2)	3	1	3 (1+2)	3
Drugie uruchomienie programu						
0	0	(0+0)	0	3	3 (3+0)	3
1	0	1 (0+1)	1	3	4 (3+1)	4
2	1	3 (1+2)	3	4	6 (4+2)	6

Kilka lewych terminali – dostęp do historii

- Rejestr można rozszerzyć o kolejne lewe terminale.
- Uzyskujemy wtedy wartości z wcześniejszych iteracji: x_{i-1}, x_{i-2}, x_{i-3}, ...
- Zastosowania: proste filtry, różnica kolejnych próbek, wykrywanie trendu.
- Nie jest to zamiennik dużego bufora – do wielu próbek lepsza jest tablica.



i	result i-1	result i-2	result i-3	prawy terminal
0	0	0	0	0 (0+0)
1	0	0	0	1 (0+1)
2	1	0	0	2 (0+2)
3	2	1	0	3 (0+3)
4	3	2	1	5 (1+4)
5	5	3	2	7 (2+5)

Rejestr czy auto-indexing?

	Rejestr przesuwny	Auto-indexing
Co przechowuje?	bieżący dostęp do wartości z poprzednich iteracji pętli	serię wartości tworzonych/odczytywanych w pętli zwracanych po zakończeniu pętli
Typowy cel	pamięć stanu, akumulator, poprzednia próbka	przetwarzanie tablicy / budowa tablicy
Dostęp w trakcie pętli	tak, wartość z poprzedniej iteracji	pełna tablica dopiero po zakończeniu pętli
Przykład	licznik, stan automatu	filtracja 100 pomiarów

Czas na podsumowanie tej części wykładu!

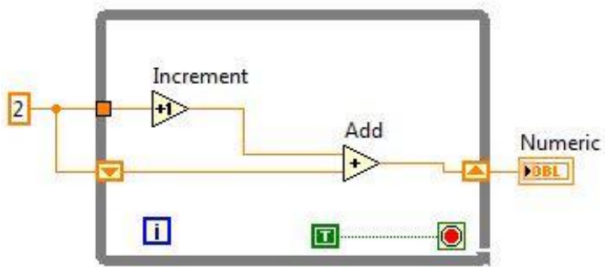
Zamiast tekstu – seria pytań kontrolnych.



Pytanie kontrolne

Jaka wartość zostanie wyświetlona w zmiennej Numeric?

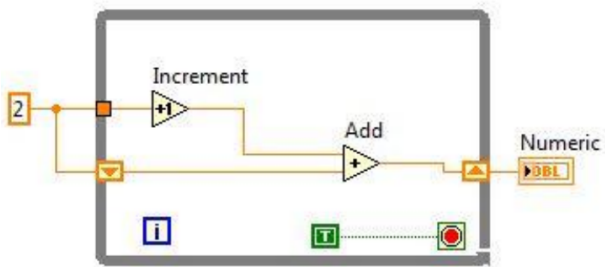
a) 0
b) 4
c) 5
d) Pętla będzie wykonywała się w nieskończoność



Pytanie kontrolne

Jaka wartość zostanie wyświetlona w zmiennej Numeric?

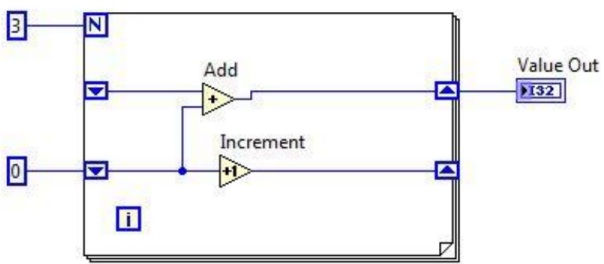
a) 0
b) 4
c) **5**
d) Pętla będzie wykonywała się w nieskończoność



Pytanie kontrolne

VI został otwarty i uruchomiony dwa razy bez zamykania i dokonywania zmian. Jaka wartość będzie wyświetlona w zmiennej Value Out po drugim uruchomieniu?

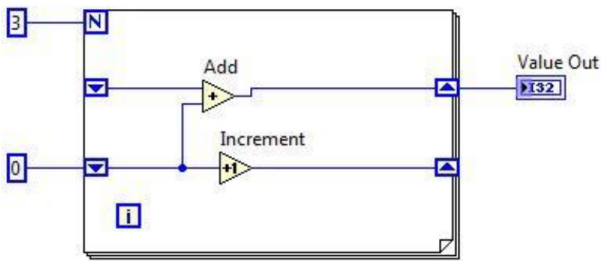
a) 3
b) 4
c) 5
d) 6



Pytanie kontrolne

VI został otwarty i uruchomiony dwa razy bez zamykania i dokonywania zmian. Jaka wartość będzie wyświetlona w zmiennej Value Out po drugim uruchomieniu?

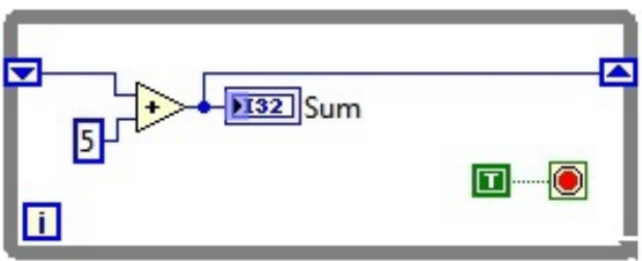
a) 3
b) 4
c) 5
d) **6**



Pytanie kontrolne

Poniższy subVI został załadowany do pamięci i uruchomiony. Po pewnym czasie ten sam subVI wciąż znajduje się w pamięci i został uruchomiony drugi raz. Jaka wartość zostanie przekazana do zmiennej Sum po uruchomieniu subVI drugi raz?

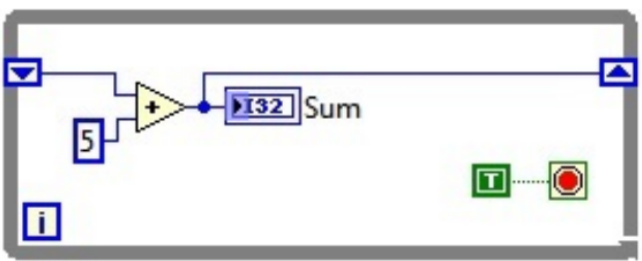
a) 5
b) 10
c) Nie wiadomo
d) 0



Pytanie kontrolne

Poniższy subVI został załadowany do pamięci i uruchomiony. Po pewnym czasie ten sam subVI wciąż znajduje się w pamięci i został uruchomiony drugi raz. Jaka wartość zostanie przekazana do zmiennej Sum po uruchomieniu subVI drugi raz?

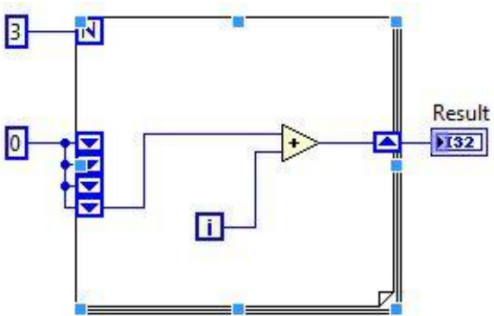
a) 5
b) **10**
c) Nie wiadomo
d) 0



Pytanie kontrolne

Jaka wartość zostanie wyświetlona w zmiennej Result po zakończeniu działania VI?

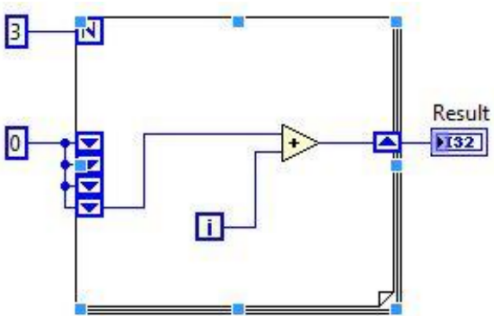
a) 3
b) 2
c) 1
d) 0



Pytanie kontrolne

Jaka wartość zostanie wyświetlona w zmiennej Result po zakończeniu działania VI?

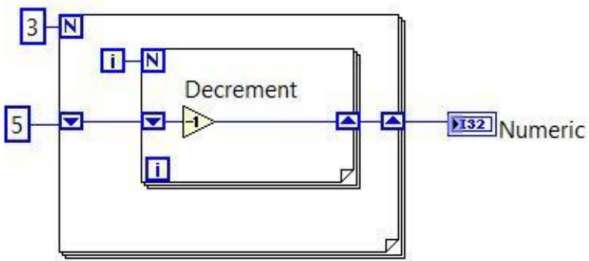
a) 3
b) **2**
c) 1
d) 0



Pytanie kontrolne

Jaka wartość zostanie wyświetlona w zmiennej Numeric?

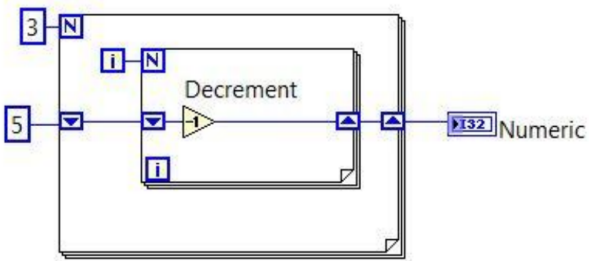
a) 0
b) 1
c) 2
d) 4



Pytanie kontrolne

Jaka wartość zostanie wyświetlona w zmiennej Numeric?

a) 0
b) 1
c) **2**
d) 4



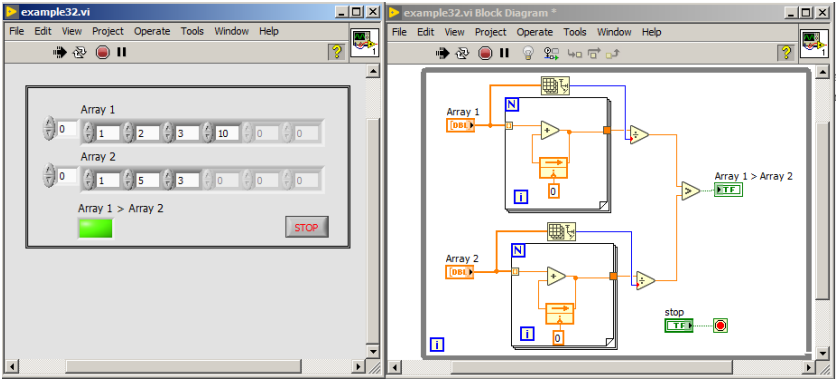
2. SubVI

Jak podzielić większy program na czytelne, wielokrotnie używane bloki?



Po co nam SubVI?

- Powtarzający się fragment kodu powinien zostać wydzielony do osobnego VI.
- SubVI pełni rolę podobną do **funkcji** w językach tekstowych.
- Dobrze zaprojektowany SubVI ma jasno określone wejścia, wyjścia i jedno zadanie.



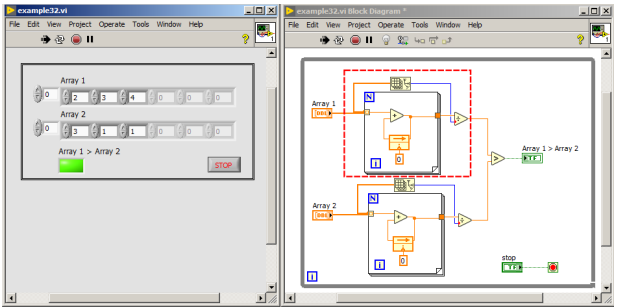
Dwa sposoby tworzenia SubVI

Od początku

Tworzymy nowy VI, definiujemy jego funkcję, Front Panel, connector pane i ikonę. To zwykle najlepsze rozwiązanie dla świadomie projektowanego modułu.

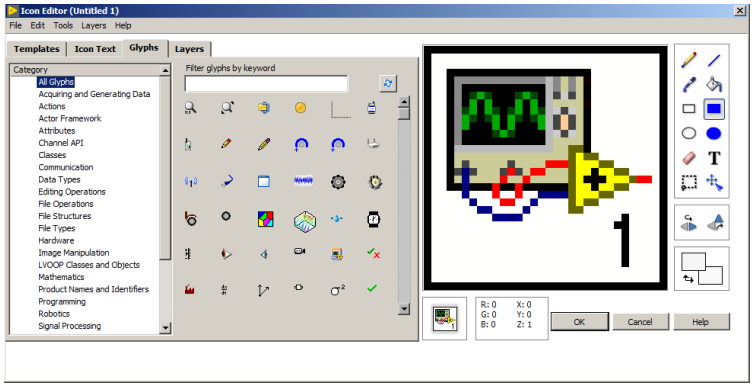
Z istniejącego kodu

Zaznaczamy fragment Block Diagram i wybieramy **Edit** → **Create SubVI**. LabVIEW automatycznie tworzy wejścia i wyjścia wynikające z zaznaczenia.

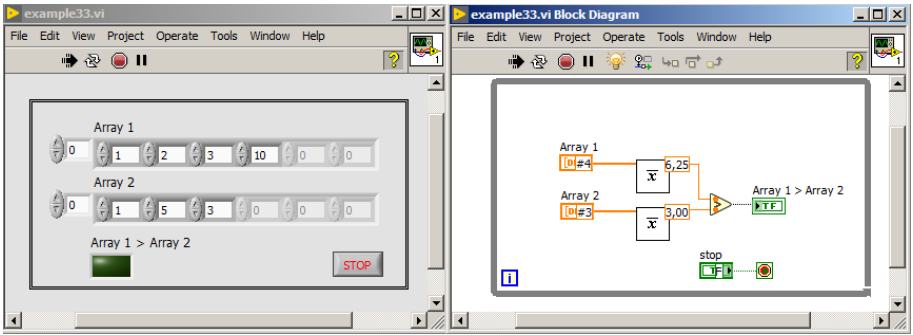


Ikona SubVI – powinna wskazywać na funkcjonalność subVI

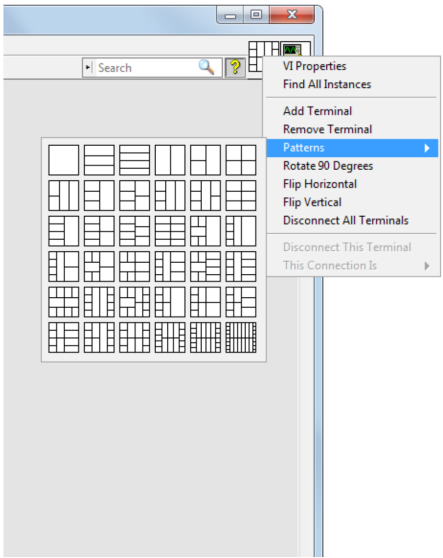
- Domyślna ikona nie opisuje funkcjonalności.
- W większej aplikacji kilka identycznych ikon szybko zamienia diagram w zagadkę.
- Dobra ikona powinna być prosta, kontrastowa i rozpoznawalna przy małym rozmiarze.
- Krótkie oznaczenie tekstowe + symbol często wystarcza.



Zmodyfikowany przykład wykorzystujący subVI



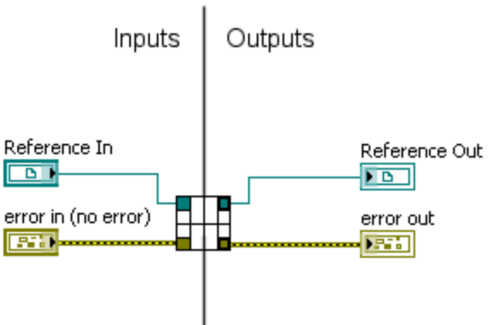
Connector Pane – interfejs SubVI



- Connector Pane określa rozmieszczenie terminali na ikonie SubVI.
- Reprezentuje wszystkie możliwe połączenia.
- Tworząc subVI możesz wybrać wzór rozłożenia terminali. Zalecany wzór to: 4x2x2x4 (lewy bok, góra, dół, prawy bok).
- Każdy terminal może być wymagany do podłączenia, rekomendowany lub opcjonalny. Jedną z powyższych opcji wybiera się poprzez kliknięcie na terminal na connector pane i wybranie**This Connection Is...**.

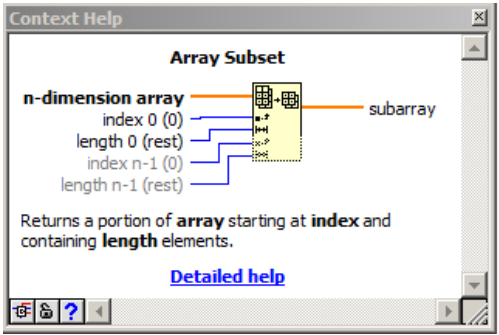
Konwencja rozmieszczenia terminali

- wejścia typowo po **lewej**, wyjścia po **prawej**,
- główny obiekt/referencja wysoko,
- error in / error out zwykle na dole – do szczegółów wrócimy przy pracy ze sprzętem,
- podobne SubVI powinny mieć podobny układ terminali.



Required, Recommended, Optional

- **Required** – brak połączenia uniemożliwia uruchomienie VI. W pomocy kontekstowej zaznaczony pogrubioną czcionką.
- **Recommended** – zwykle powinno być podłączone; brak połączenia jednak umożliwia wykorzystanie funkcji/subVI. W pomocy kontekstowej zaznaczony standardową czcionką.
- **Optional** – podanie wartości nie jest wymagane i zazwyczaj dotyczy jakieś dodatkowej funkcjonalności funkcji/subVI. W pomocy kontekstowej zaznaczony wyszarzałą czcionką.



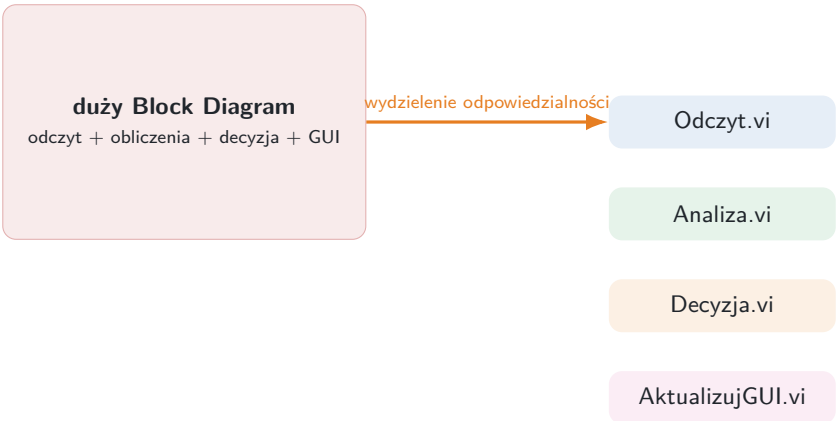
SubVI a dataflow

- SubVI jest zwykłym węzłem dataflow: uruchamia się, gdy dostępne są wszystkie wymagane dane wejściowe.
- Wyniki są dostępne dopiero po zakończeniu jego wykonania.
- Dwa niezależne SubVI mogą wykonywać się równolegle, jeśli nie ma między nimi zależności danych.
- Aby wymusić kolejność, najlepiej przekazać klaster błędów - więcej na ten temat na kolejnym wykładzie.

Pamiętaj

Położenie ikon na Block Diagramie nie określa kolejności wykonania.

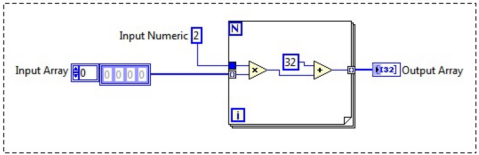
Refaktoryzacja: z dużego diagramu do modułów



Korzyść: łatwiej czytać, testować, poprawiać i ponownie wykorzystywać kod.

Pytanie kontrolne

Po wybraniu opcji *Create SubVI* z menu Edycji, jaki rezultat otrzymasz?



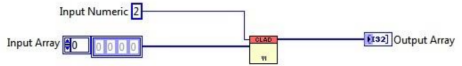
A



B



C

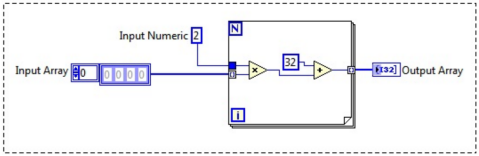


D



Pytanie kontrolne

Po wybraniu opcji *Create SubVI* z menu Edycji, jaki rezultat otrzymasz?



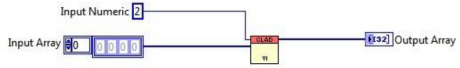
A



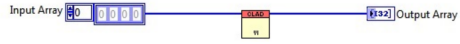
B



C



D

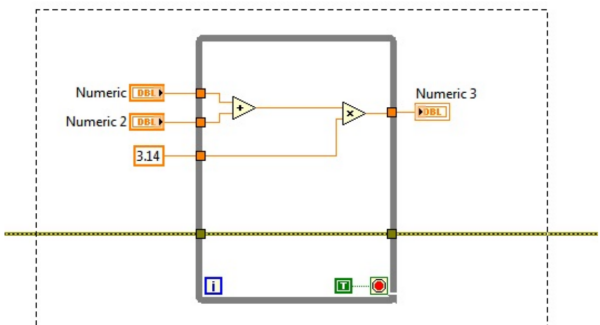


Poprawna odpowiedź: B

Pytanie kontrolne

SubVI został stworzony z zaznaczonego obszaru kodu. Ile terminali posiada subVI?

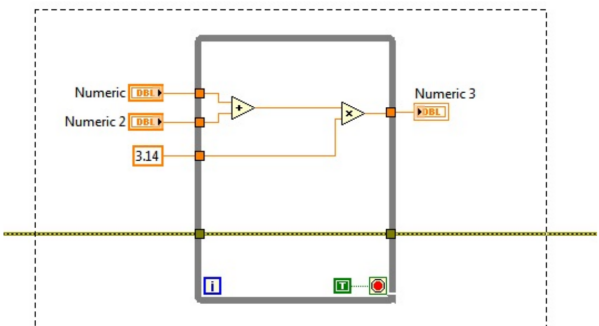
a) 3
b) 4
c) 5
d) 6



Pytanie kontrolne

SubVI został stworzony z zaznaczonego obszaru kodu. Ile terminali posiada subVI?

a) 3
b) 4
c) **5**
d) 6



Pytanie kontrolne

Jeśli nazwa danej wejściowej jest oznaczona pogrubioną czcionką w Show Context Help, to oznacza to że:

a) Dana wejściowa musi być skalarem.
b) Dana wejściowa jest zalecana ale nie wymagana.
c) Dana wejściowa jest wymagana.
d) Strzałka run zostanie przekreślona jeśli dana wejściowa nie będzie podłączona.

Pytanie kontrolne

Jeśli nazwa danej wejściowej jest oznaczona pogrubioną czcionką w Show Context Help, to oznacza to że:

a) Dana wejściowa musi być skalarem.

b) Dana wejściowa jest zalecana ale nie wymagana.

c) **Dana wejściowa jest wymagana.**

d) **Strzałka run zostanie przekreślona jeśli dana wejściowa nie będzie podłączona.**

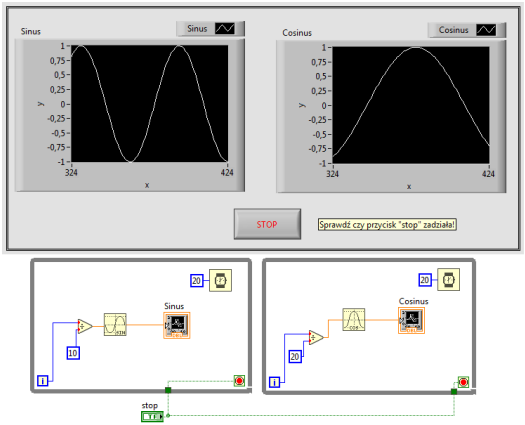
3. Zmienne lokalne

Kiedy potrzebujemy dostępu do kontrolki poza jej terminalem?



Komunikacja między dwiema pętlami

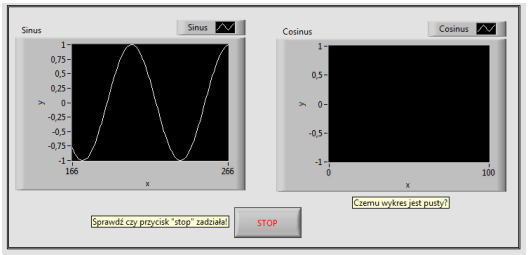
W jaki sposób zatrzymać dwie pętle Loop w tym samym czasie?
(Sposób niepoprawny)



1_rownolegle_petle_wersja1.vi

Komunikacja między dwiema pętlami

W jaki sposób zatrzymać dwie pętle Loop w tym samym czasie?
(Sposób niepoprawny)

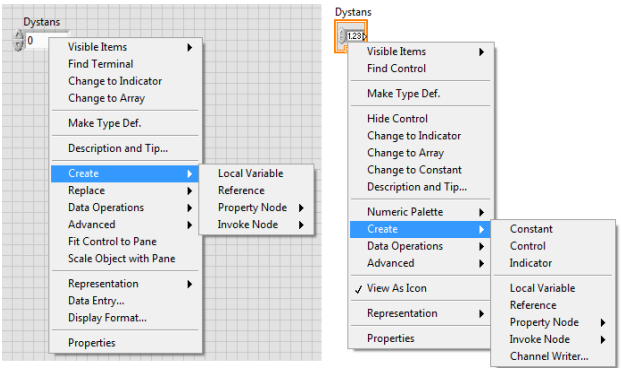


1_rownolegle_petle_wersja2.vi

Zmienne lokalne

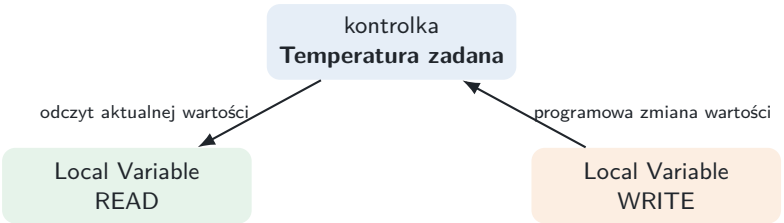
- Zmienne lokalne są stosowane do przekazywania danych w ramach jednego VI bez konieczności używania bezpośrednich połączeń. Dane są zapisywane w kontrolkach/indykatorach na front panel-u.
- Standardowo z kontrolek możemy tylko odczytywać dane a do indykatorów zapisywać. Zmienne lokalne umożliwiają zapis i odczyt do/z kontrolki jak i indykatora.
- Wykorzystanie: komunikacja między równoległymi pętlami, zapis danych w zależności od sytuacji (np. struktura case) oraz inicjalizacja kontrolek/indykatorów.
- Zmienne lokalne mogą być używane jedynie w ramach jednego subVI.
- Ich działanie jest wolniejsze niż terminali.
- Kontrolki logiczne muszą być typu switch.

Tworzenie zmiennej lokalnej



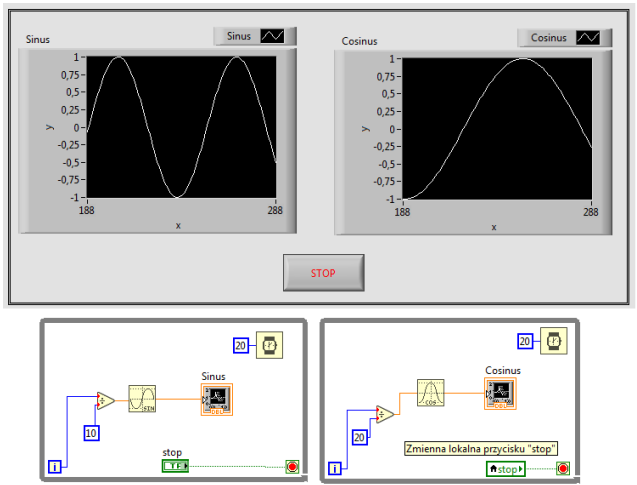
- prawy przycisk na kontrolce/terminalu,
- *Create* → *Local Variable*,
- lokalna zmienna odwołuje się do elementu z tego samego VI,
- można przełączyć kierunek: *Change to Read* / *Change to Write*.

Read i Write – dwie różne operacje



- Read pobiera aktualną wartość kontrolki/indykatora.
- Write ustawia nową wartość elementu Front Panelu z poziomu programu.

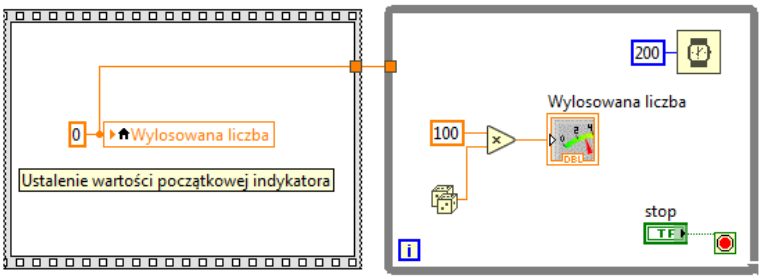
Komunikacja między dwiema pętlami - rozwiązanie



1_rownolegle_petle_wersja3.vi

Przydatne zastosowania w miniprojekcie

- ustawienie domyślnej wartości kontrolki podczas inicjalizacji (patrz. kod poniżej),
- ustawienie różnych wartości indykatora w zależności od warunku (struktura *case*),
- komunikacja między równoległymi pętlami (uwaga: koniecznie muszą pracować z tą samą częstotliwością),
- aktualizacja wskaźnika z kilku stanów maszyny (więcej na ten temat w dalszej części wykładu).



Zmienne lokalne omijają standardowy *dataflow*

Przewód

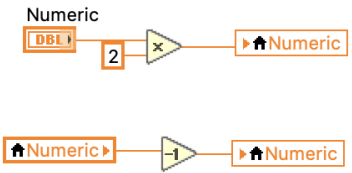
Tworzy jawną zależność. Odbiorca nie może wykonać się przed pojawieniem się danych.

Zmienna lokalna

Odczyt lub zapis może wykonać się wtedy, gdy jego inne zależności są spełnione. Sama wspólna nazwa nie ustala kolejności.

Konsekwencja

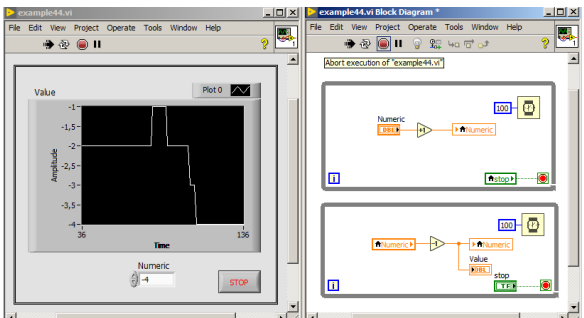
Dwa równoległe fragmenty zapisujące lub odczytujące ten sam element mogą prowadzić do wyniku zależnego od kolejności wykonania.



Możliwe wyniki:
1. Numeric = Numeric*2 -1
2. Numeric = (Numeric-1)*2

Race condition – dlaczego wynik może się zmieniać?

- Race condition powstaje, gdy wynik zależy od tego, który z równoległych fragmentów kodu wykona się pierwszy.
- Kolejność może różnić się między uruchomieniami.
- Test „działało 100 razy” nie gwarantuje poprawności architektury.



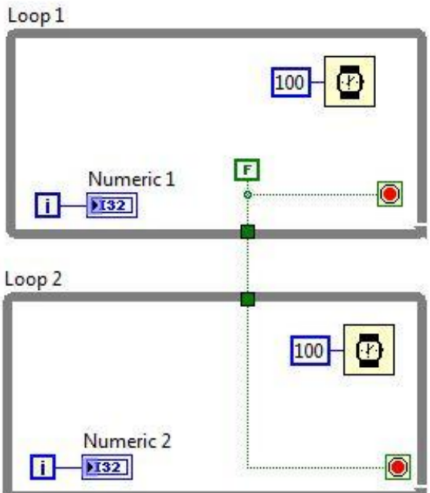
Bezpieczne zasady używania zmiennych lokalnych

- Preferuj przewody, gdy dane naturalnie płyną z jednego miejsca do drugiego.
- Dla jednego elementu staraj się mieć **jedno miejsce zapisu**; wiele miejsc odczytu nie stwarza ryzyka wystąpienia *race conditions*.
- Nie używaj lokalnej zmiennej jako mechanizmu synchronizacji.
- Nie buduj na niej „ukrytej" kolejności wykonywania programu.
- W miniprojekcie: zmienna lokalna jest dobra do odczytu wartości i prostych resetów, ale operacje zapisów realizuj w inny sposób np. korzystając z rejestru przesuwnego.

Pytanie kontrolne

Jak będzie działał poniższy program?

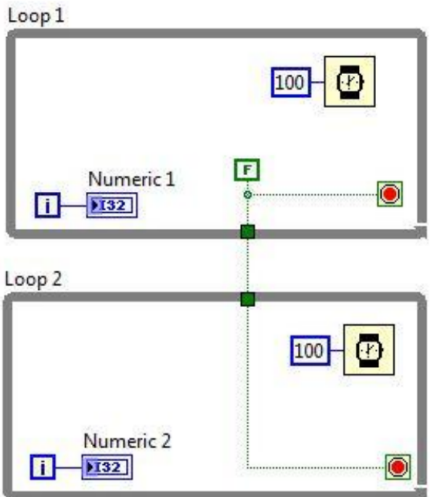
a) Pętla 1 i 2 będą wykonywać się równolegle.
b) Oby dwie pętle uruchomią się tylko jeden raz.
c) Pętla 2 będzie wykonywać się po zatrzymaniu działania pętli 1.
d) Pętla 1 będzie działała w nieskończoność a pętla 2 nigdy się nie wykona.



Pytanie kontrolne

Jak będzie działał poniższy program?

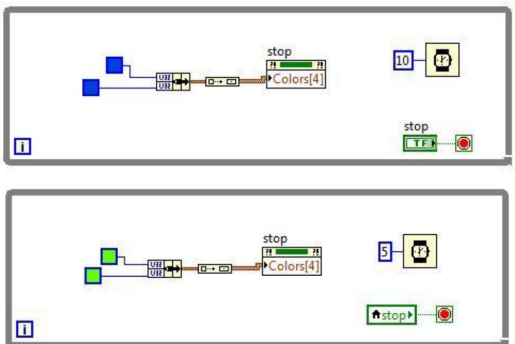
a) Pętla 1 i 2 będą wykonywać się równolegle.
b) Oby dwie pętle uruchomią się tylko jeden raz.
c) Pętla 2 będzie wykonywać się po zatrzymaniu działania pętli 1.
d) **Pętla 1 będzie działała w nieskończoność a pętla 2 nigdy się nie wykona.**



Pytanie kontrolne

Poniższy kawałek kodu wykonywał się 11 ms. Jaki kolor będzie miał przycisk stop po zakończeniu programu?

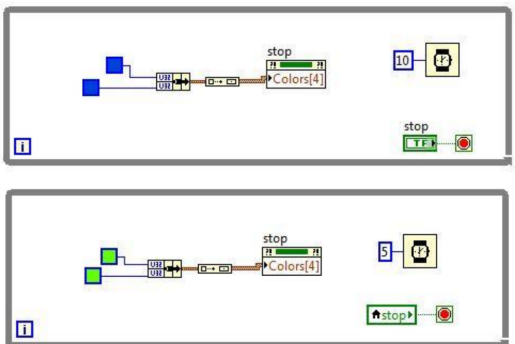
a) Niebieski
b) Zielony
c) Zielono-niebieski
d) Nie wiadomo



Pytanie kontrolne

Poniższy kawałek kodu wykonywał się 11 ms. Jaki kolor będzie miał przycisk stop po zakończeniu programu?

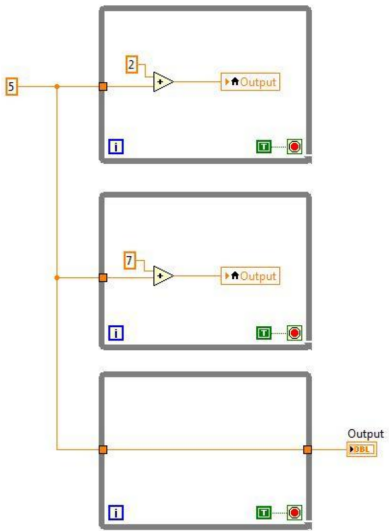
a) Niebieski
b) Zielony
c) Zielono-niebieski
d) **Nie wiadomo**



Pytanie kontrolne

Jaka wartość zostanie wyświetlona w *Output* po zakończeniu działania programu?

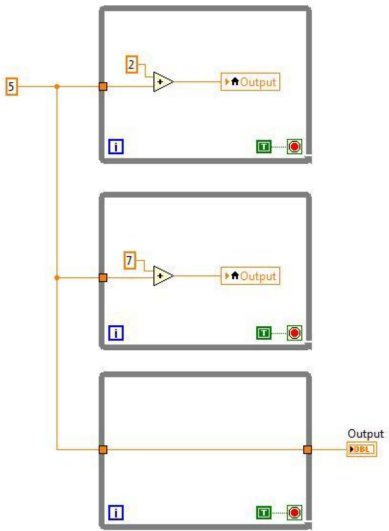
a) 5
b) 7
c) 12
d) Nie wiadomo



Pytanie kontrolne

Jaka wartość zostanie wyświetlona w *Output* po zakończeniu działania programu?

a) 5
b) 7
c) 12
d) **Nie wiadomo**



Pytanie kontrolne

Jeśli potrzebujesz zmienić programistycznie wartość w Numeric Control, jaka strategia będzie najlepsza?

a) Poprzez użycie zmiennej funkcyjnej.
b) Poprzez użycie zmiennej lokalnej.
c) Poprzez ustawienie wartości domyślnej.
d) Poprzez użycie referencji.

Pytanie kontrolne

Jeśli potrzebujesz zmienić programistycznie wartość w Numeric Control, jaka strategia będzie najlepsza?

a) Poprzez użycie zmiennej funkcyjnej.
b) **Poprzez użycie zmiennej lokalnej.**
c) Poprzez ustawienie wartości domyślnej.
d) Poprzez użycie referencji.

4. Event Structure

Zamiast ciągle pytać: reaguj dopiero, gdy wydarzy się coś istotnego



Polling czy reakcja na zdarzenie?

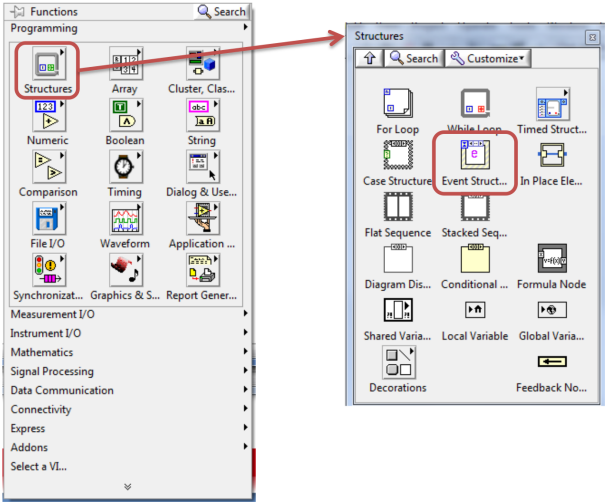
Polling

- pętla co chwilę sprawdza stan przycisku,
- procesor jest obciążony powtarzaniem obliczeń z tymi samymi wartościami.

Event-driven

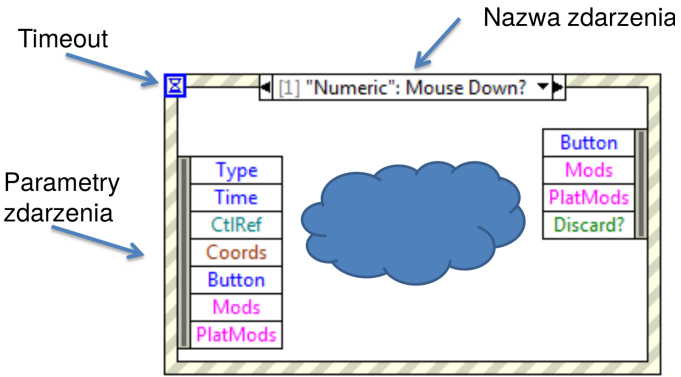
- program czeka na zdarzenie,
- odpowiednia ramka wykonuje się dopiero po jego wystąpieniu,
- naturalny model dla interfejsu użytkownika.

Struktura zdarzeniowa

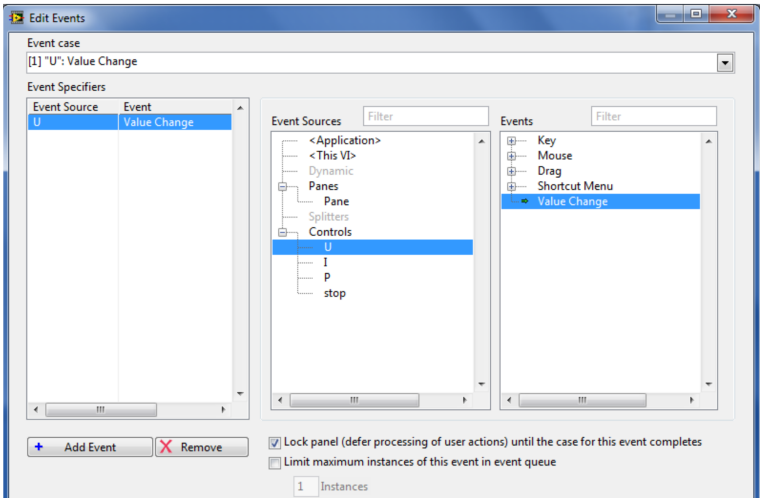


Struktura zdarzeniowa

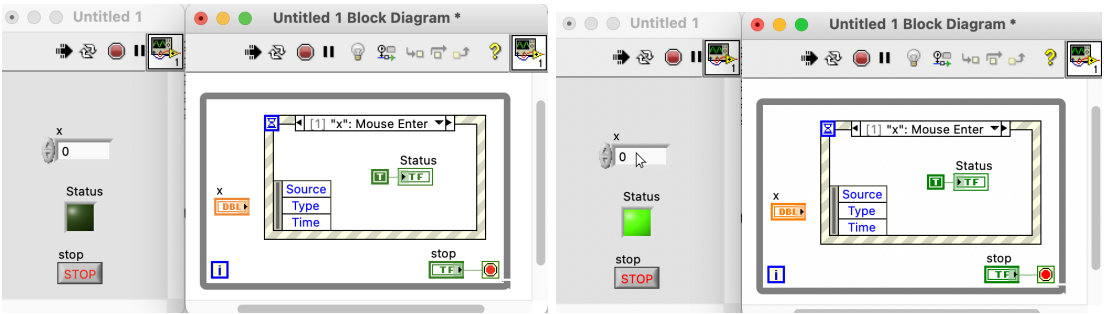
- Umożliwia wykonanie danego fragmentu kodu w zależności do zdarzenia.
- Przykładowe zdarzenia to: użytkownik najechał na wykres, użytkownik zmienił wartość kontrolki itd.
- Nazwa zdarzenia ma format: *nazwa elementu związanego z zdarzeniem : co wywołuje zdarzenie*.



Struktura zdarzeniowa



Przykład



Przyklad5.vi

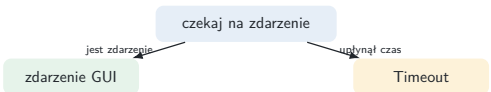
Value Change – podstawowe zdarzenie GUI

- Zdarzenie **Value Change** pojawia się, gdy wartość kontrolki zostanie zmieniona przez użytkownika.
- Najczęstsze zastosowania: START, STOP, RESET, wybór trybu, zmiana nastawy.
- W jednej ramce można obsługiwać jedno lub kilka zdarzeń, ale kod powinien pozostać czytelny.
- Zdarzenie nie jest pętlą – dlatego Event Structure typowo umieszczamy **wewnątrz While Loop**.

START: Value Change STOP: Value Change TRYB: Value Change

Timeout – zdarzenie, gdy nic się nie wydarzyło

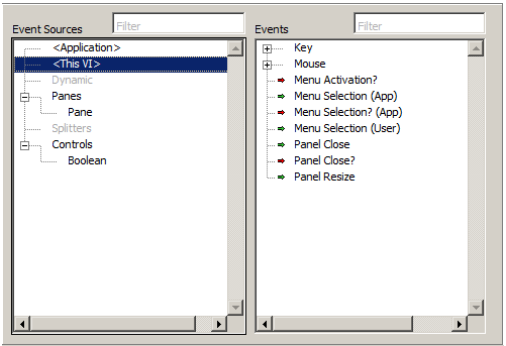
- Terminal Timeout określa maksymalny czas oczekiwania na zdarzenie.
- Po upływie czasu wykonywana jest ramka *Timeout*.
- Wartość -1 oznacza oczekiwanie bez limitu.
- Timeout jest wygodny do prostych czynności okresowych: odświeżenie wskaźnika, sprawdzenie warunku, licznik czasu.



Notify / Filter Events

Zdarzenia dzielimy na:

- Notify Events (zielone strzałki) - po wystąpieniu zdarzenia zostanie wykonana odpowiednia ramka struktury zdarzeniowej,
- Filter Events (czerwone strzałki) - domyślna akcja dla danego zdarzenia się nie wykona lecz zostanie wywołana dana ramka struktury zdarzeniowej np. nie zostanie zamknięty VI po naciśnięciu "Xłecz zostanie wywołany kod z struktury zdarzeniowej.



Typowe pułapki Event Structure

- umieszczenie Event Structure poza pętlą – obsłuży zdarzenie tylko w przebiegu programu i VI zakończy się,
- kilka niezależnych Event Structure próbujących obsługiwać ten sam interfejs,
- bardzo długi kod wewnątrz ramki,
- brak sensownego sposobu zakończenia pętli,
- mechaniczna akcja przycisku niedopasowana do sposobu odczytu,
- traktowanie Timeout jak precyzyjnego zegara czasu rzeczywistego.

Pytanie kontrolne

Co jest główną zaletą Event Structure w interaktywnym GUI?

(a) ciągłe odpytywanie wszystkich kontrolek
(b) automatyczne tworzenie plików
(c) zastąpienie wszystkich pętli For
(d) wykonywanie kodu dopiero po określonym zdarzeniu

Event Structure

Pytanie kontrolne

Co jest główną zaletą Event Structure w interaktywnym GUI?

(a) ciągłe odpytywanie wszystkich kontrolek
(b) automatyczne tworzenie plików
(c) zastąpienie wszystkich pętli For
(d) **wykonywanie kodu dopiero po określonym zdarzeniu**

Event Structure

Pytanie kontrolne

Dlaczego nie warto umieszczać długiego pomiaru bezpośrednio w ramce Event Structure?

(a) bo Event Structure nie obsługuje liczb DBL
(b) bo podczas długiej obsługi może nie reagować na kolejne zdarzenia użytkownika
(c) bo wykres nie może być w pętli
(d) bo SubVI nie działa wewnątrz Event

Event Structure

Pytanie kontrolne

Dlaczego nie warto umieszczać długiego pomiaru bezpośrednio w ramce Event Structure?

(a) bo Event Structure nie obsługuje liczb DBL
(b) **bo podczas długiej obsługi może nie reagować na kolejne zdarzenia użytkownika**
(c) bo wykres nie może być w pętli
(d) bo SubVI nie działa wewnątrz Event

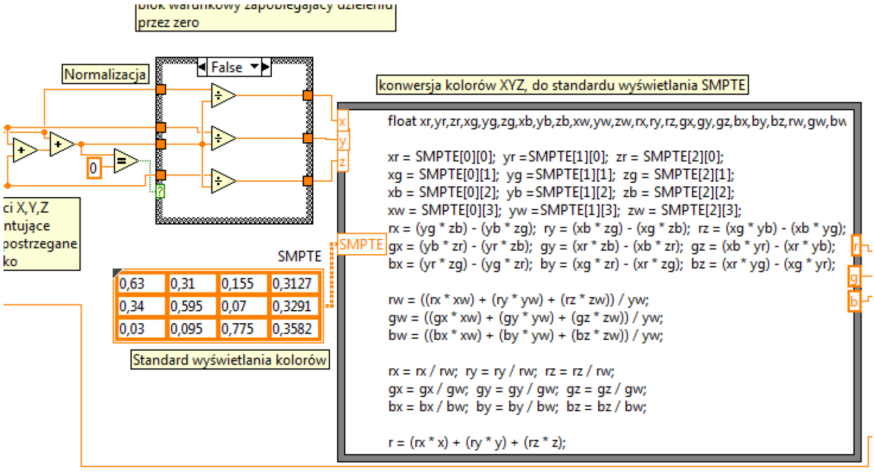
Event Structure

5. Maszyna stanów

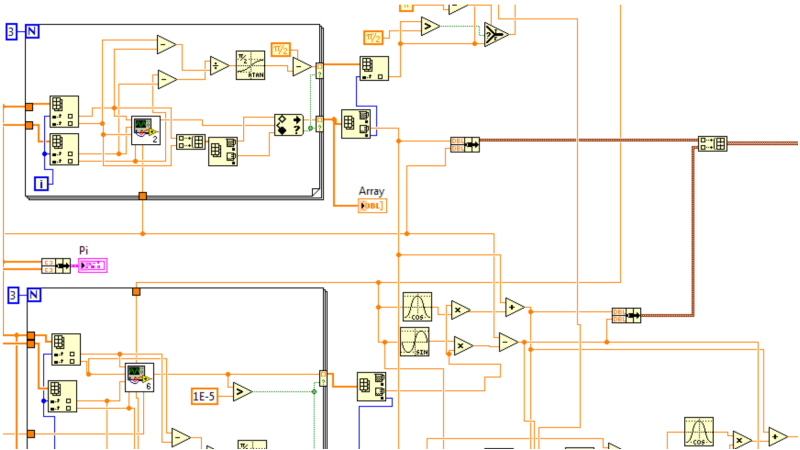
Jak podzielić działanie aplikacji na czytelne kroki?



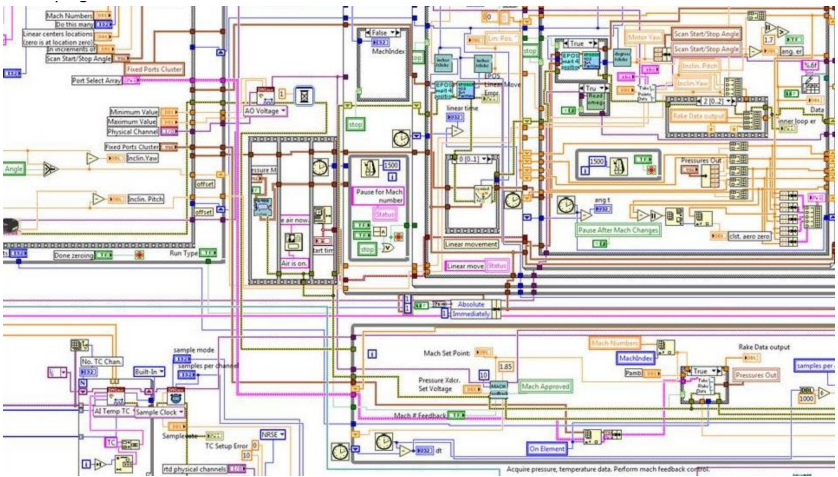
Przykłady złych praktyk → *Spaghetti code*



Przykłady złych praktyk → *Spaghetti code*



Przykłady złych praktyk → *Spaghetti code*



Dlaczego warto stosować wzorce programistyczne?

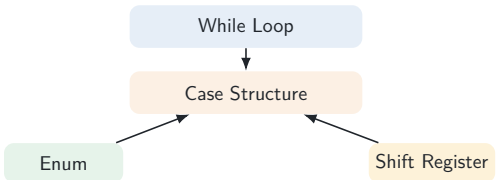
- **Wzorzec programistyczny** to sposób implementowania kodu programu mający na celu rozwiązanie konkretnych problemów programistycznych.
- **Dlaczego warto stosować wzorce?**
  - Gotowe rozwiązania - szablony zapewniające optymalną architekturę programu,
  - Spełniają kryteria dobrego programowania (skalowalne, czytelne, modułowe).
- Najpopularniejszy wzorzec programistyczny to **Maszyna stanów**:
  - Dobra do aplikacji, która ma kilka etapów, np. oczekiwanie, konfigurację, pomiar, pauzę i zakończenie.
  - Każdy etap wykonuje inny fragment kodu.
  - Po zakończeniu etapu trzeba zdecydować, **co ma wykonać się następnie**.

Maszyna stanów

Modelujemy aplikację jako zbiór nazwanych stanów oraz reguł przejścia z jednego stanu do drugiego.

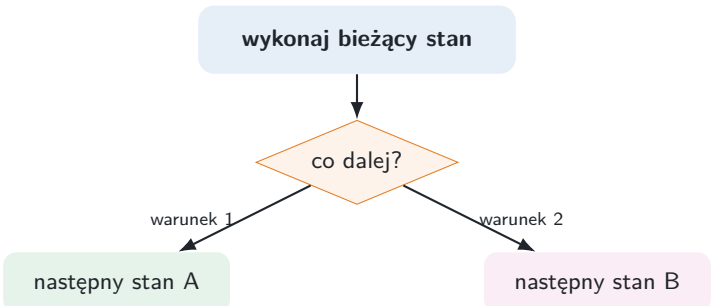
Klasyczna maszyna stanów w LabVIEW

- **While Loop** – powtarza wykonywanie kolejnych stanów,
- **Case Structure** – zawiera kod każdego stanu,
- **Enum** – opisuje nazwy stanów,
- **Shift Register** – przechowuje informację o kolejnym stanie.



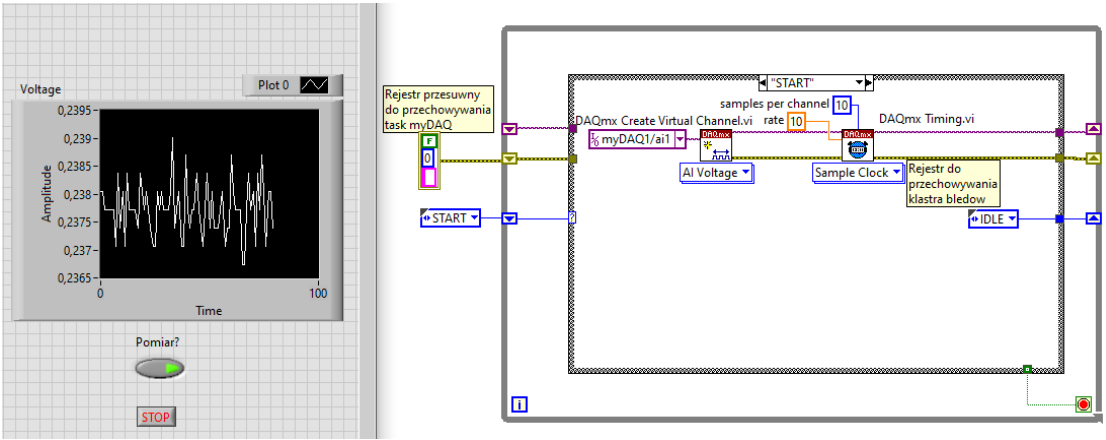
Dobry stan i najważniejsze pytanie: co dalej?

- Stan powinien mieć czytelną nazwę, wykonywać jedno spójne zadanie i na końcu wskazywać kolejny stan.
- Nie próbujemy realizować całej aplikacji w jednej ramce Case.

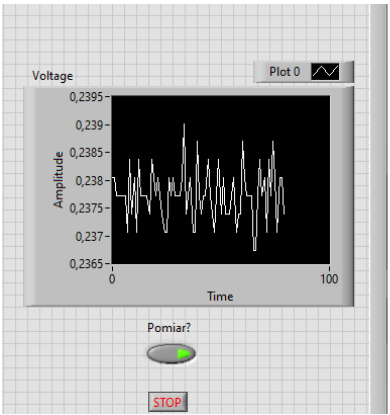


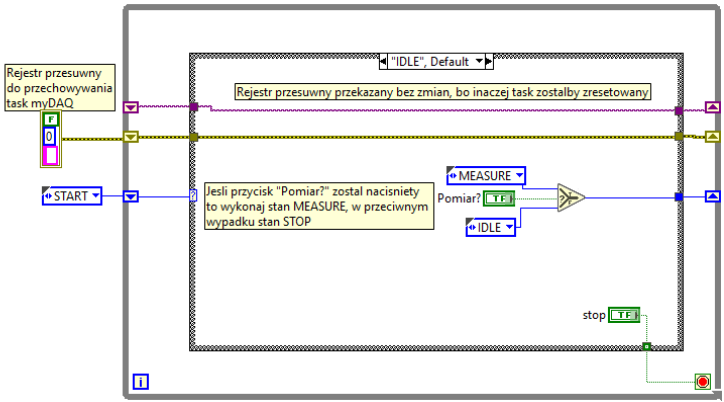
Wynik tej decyzji wpisujemy do prawego terminala rejestru przesuwnego.

Przykład przewodni: stanowisko pomiarowe

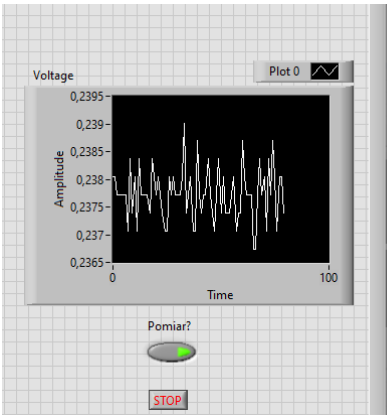


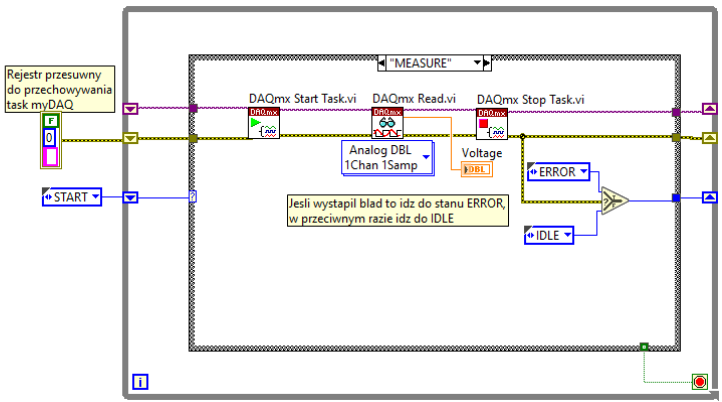
Przykład przewodni: stanowisko pomiarowe



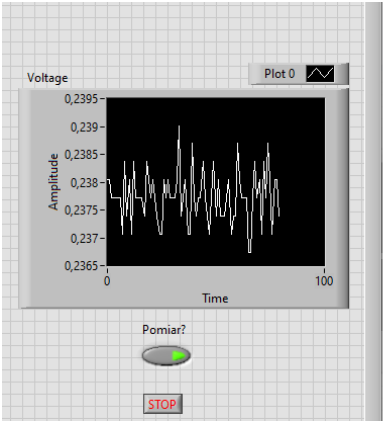


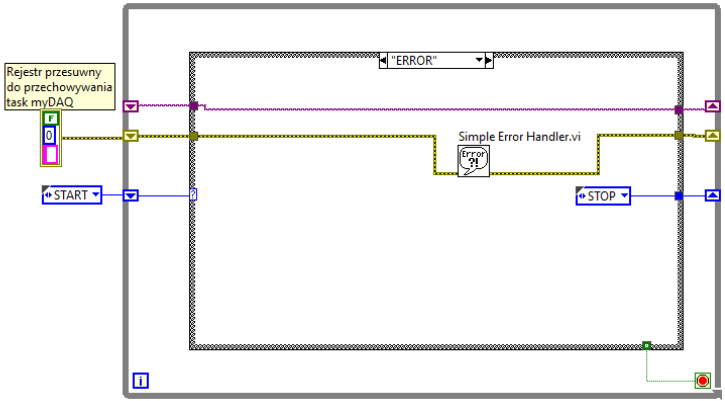
Przykład przewodni: stanowisko pomiarowe



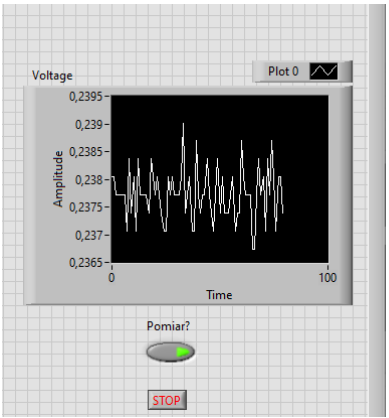


Przykład przewodni: stanowisko pomiarowe





Przykład przewodni: stanowisko pomiarowe



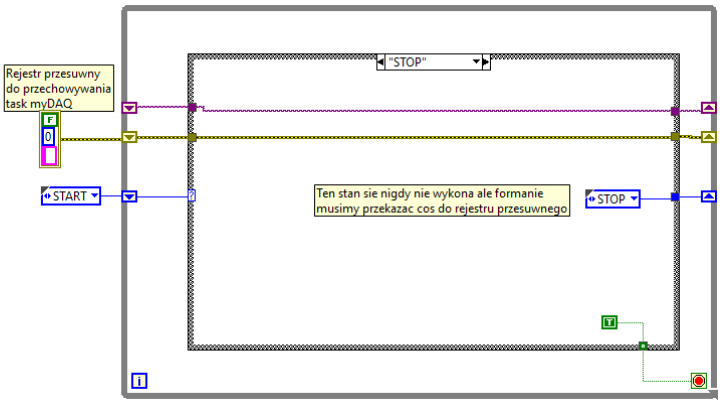
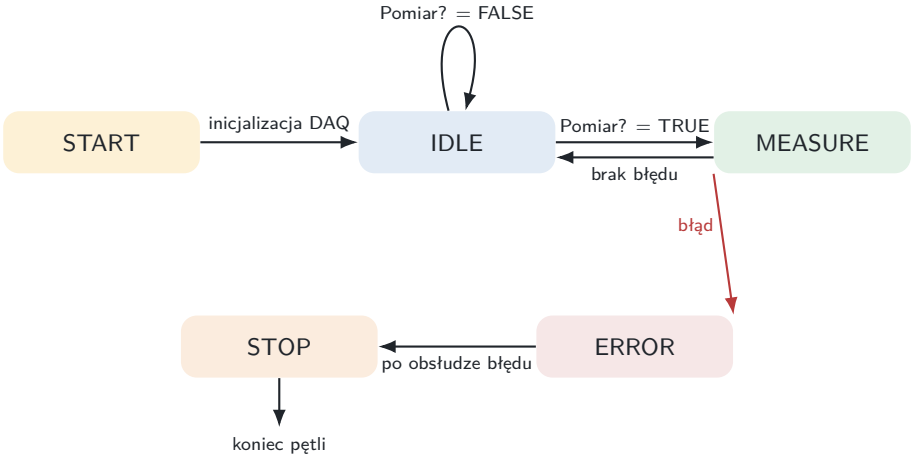


Diagram przejść stanów



W stanie **IDLE** przycisk **STOP** może również zakończyć pętlę bez przechodzenia przez stan STOP. Stan STOP jest wykorzystywany w pokazanym kodzie po obsłudze stanu ERROR.

Tabela przejść – prosty sposób planowania przed kodowaniem

Stan	Warunek / zdarzenie	Następny stan
START	Uruchomienie programu	IDLE
IDLE	Kliknięcie na *Pomiar?*	MEASURE
MEASURE	brak błędu	IDLE
MEASURE	błąd	ERROR
ERROR	-	STOP

Najpierw projektujemy stany i przejścia, dopiero potem rysujemy kod LabVIEW.

Gdzie w maszynie stanów przydają się SubVI i zmienne lokalne?

SubVI

- większe operacje w stanach,
- obliczenia wielokrotnie używane,
- konfiguracja, pojedynczy pomiar, aktualizacja algorytmu.

Zmienne lokalne

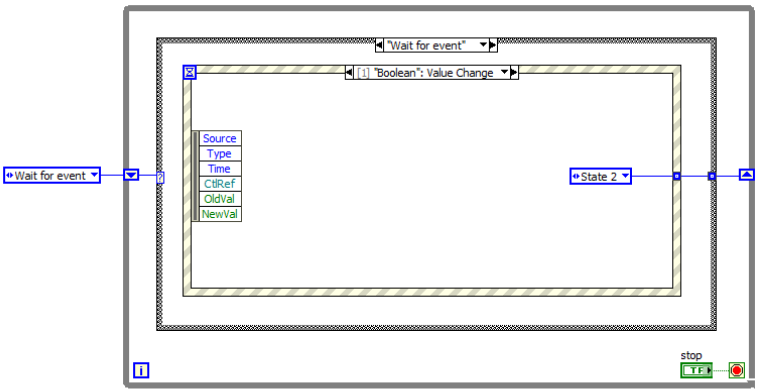
- reset kontrolki/wskaźnika,
- prosta aktualizacja GUI z różnych stanów.

Ale stan aplikacji

Bieżący / następny stan oraz wyniki przechowujemy w **rejestrze przesuwnym**, nie w zmiennej lokalnej.

Event Structure + maszyna stanów

- Event Structure umożliwia reakcję na wybory użytkownika.
- W prostym projekcie można umieścić Event Structure tylko w stanie oczekiwania / obsługi GUI.
- Zdarzenie ustawia następny stan, a kod stanu wykonuje właściwe zadanie.



Maszyna stanów – mocne strony i ograniczenia

Zalety

- czytelny podział na etapy,
- łatwa rozbudowa o kolejne stany,
- naturalne miejsce na inicjalizację, pracę, pauzę i błąd,
- dobra architektura dla małych i średnich projektów.

Ograniczenia

- jeden długi stan może blokować reakcję GUI,
- brak obsługi równoległych zadań,
- zbyt wiele przejść tworzy trudny „labirynt".

Jak zaplanować miniprojekt przed otwarciem LabVIEW?

1. Narysuj diagram stanów na kartce.
2. Wypisz wszystkie stany jako wartości Enum TypeDef.
3. Dla każdego stanu zapisz: **co robi?** oraz **dokąd może przejść?**
4. Określ, które dane muszą być pamiętane w rejestrach przesuwnych.
5. Wydziel większe operacje do SubVI.
6. Zdecyduj, które przyciski obsługuje Event Structure.
7. Dopiero wtedy zbuduj While + Case + Shift Register.

Czas na podsumowanie tej części wykładu!

Zamiast tekstu – seria pytań kontrolnych.



Pytanie kontrolne

Który zestaw elementów tworzy klasyczną prostą maszynę stanów w LabVIEW?

(a) For Loop + XY Graph + Local Variable
(b) While Loop + Case Structure + Enum + Shift Register
(c) Event Structure + Waveform Chart
(d) Cluster + Array Size + Select

maszyna stanów

Pytanie kontrolne

Który zestaw elementów tworzy klasyczną prostą maszynę stanów w LabVIEW?

(a) For Loop + XY Graph + Local Variable
(b) **While Loop + Case Structure + Enum + Shift Register**
(c) Event Structure + Waveform Chart
(d) Cluster + Array Size + Select

maszyna stanów

Pytanie kontrolne

Gdzie najlepiej przechowywać informację o następnym stanie klasycznej maszyny stanów?

(a) w rejestrze przesuwnym pętli
(b) w nazwie kontrolki
(c) w osi X wykresu
(d) wyłącznie w zmiennej lokalnej

maszyna stanów

Pytanie kontrolne

Gdzie najlepiej przechowywać informację o następnym stanie klasycznej maszyny stanów?

(a) **w rejestrze przesuwnym pętli**
(b) w nazwie kontrolki
(c) w osi X wykresu
(d) wyłącznie w zmiennej lokalnej

maszyna stanów

Podsumowanie

Program może pamiętać, reagować i zmieniać zachowanie – ale tylko wtedy, gdy jasno zaprojektujemy przepływ danych i stanów.