Podstawy programowania w LabView (2)

Instrukcje strukturalne

W pierwszej części poznaliśmy środowisko programistyczne. W tej części poznamy podstawowe struktury języka programowania graficznego G. Sterujące wykonaniem programu i ułatwiające budowanie wygodnego interfejsu użytkownika.

W graficznym języku programowania program nie wykonuje się kolejno linia po linii, tak jak ma to miejsce w tekstowych językach programowania. O kolejności wykonania kodu decydują:

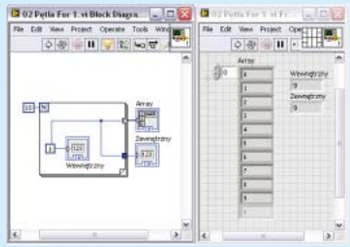
1. Kolejność przepływu danych – kod wykonuje się wtedy, gdy dostępne są wszystkie dane. Czyli wszystkie poprzedzające instrukcje dostarczające danych dla funkcji zakończyły swoje działanie.
2. Instrukcje strukturalne – jeśli funkcje nie przekazują sobie żadnych danych o kolejności wykonywania kodu, decydują instrukcje, takie jak np. pętle *For Loop*, *While Loop*, struktury *Case Structure*, *Sequence Structure* itp.
3. Kompilator – w przypadku, gdy żadne z powyższych nie ma zastosowania, o kolejności wykonania kodu decyduje kompilator i nie jesteśmy w stanie przewidzieć, która funkcja wykona się jako pierwsza, a która jako druga. Kolejność umieszczania funkcji na diagramie oraz ich ułożenie względem siebie nie mają na to wpływu.

Dwa umieszczone na diagramie programy będą wykonywane pseudorównolegle. Dzięki temu dla tego środowiska jest naturalne równoległe przetwarzanie danych. Wystarczy umieścić na diagramie dwie pętle *while*, pomiędzy którymi nie ma bezpośredniej wymiany danych (przez połączenia), a kod w nich zawarty będzie wykonywany równolegle. Przykład takiego programu przedstawia **rysunek 24**. Oczywiście, istnieją funkcje synchronizujące pętle wykonywane równolegle z różną prędkością, ale to temat na oddzielny kurs. W LabView instrukcje strukturalne sterują programem. Zgromadzone są w palecie struktur (*Structures*) i dostępnej w oknie diagramu. **Rysunek 10** przedstawia paletę struktur.

Struktury zgromadzone w palecie to:

- (*For Loop*) pętla for
- (*While Loop*) pętla while
- (*Timed Structures*) struktury uzależnień czasowych pozwalają w łatwiej i dokładniej kontrolować częstotliwość i czas wykonywania fragmentów kodu
- (*Case Structure*) struktura wyboru warunkowego
- (*Event Structure*) struktura zdarzeń.
- (*Sequence Structure*) struktura sekwencji
- (*Formula Node*) struktura pozwalająca na zapis fragmentów kodu w postaci poleceń podobnych do języka C
- (*Shared Variable*) – wspólna zmienna
- (*Local Variable*) – zmienna lokalna
- (*Global Variable*) – zmienna globalna
- (*Feedback Node*) – „Sprzężenie zwrotne”

Dodatkowe materiały na CD/FTP:
ftp://ep.com.pl, user: 20637, pass: 7430ukcs



Rysunek 11. Pętla For – tunele wyjściowe.

Pętla *for* jest znana z większości języków programowania wyższego poziomu, ale w LabView związane są z nią pewne dodatkowe właściwości dotyczące również pętli *while*. Zostaną one szczegółowo opisane na przykładzie pętli *for*, a w przypadku pętli *while* będą podane tylko różnice.

Pętla for

Pętla *for* wykonuje się zadaną liczbę razy. Decyduje o tym wartość przekazana do ikony N. Liczbę iteracji pętli możemy podać jako stałą lub wyliczać ją podczas działania programu. Możliwa jest też sytuacja, w której nie musimy jej podawać, ale o tym dalej. Wewnątrz pętli jest dostępny licznik, z którego możemy pobrać numer aktualnej iteracji. Licznik zaczyna zliczanie od 0 i dlatego zwraca wartość N-1.

Najlepiej poznać właściwości pętli w sposób praktyczny dla tego otwórzmy sobie LabView nowy, czysty dokument. Przechodzimy na diagram i klikając prawym przyciskiem myszy otwieramy paletę funkcji. Wybieramy *Programming -> Structures -> For Loop* i wstawiamy ją na nasz diagram. Tym razem robimy to nieco inaczej niż zwykle. Nie przeciągamy jej na diagram, ale po kliknięciu w ikonę pętli i zmianie wyglądu kursora przechodzimy do okna diagramu i rysujemy prostokąt obejmujący fragment kodu, który ma się wykonywać w pętli. My oczywiści nie mamy jeszcze żadnego diagramu, dlatego rysujemy niewielki prostokąt. Zwróćmy uwagę na przycisk *Run* natychmiast sygnalizuje błędy. Powodem jest brak liczby iteracji.

Do ikony „N” znajdującej się w lewym, górnym rogu musimy przekazać wartość określającą liczbę iteracji pętli. W naszym wypadku będzie to stała równa 10. W tym celu najeżdżamy na ikonę i gdy wygląd kursora zmieni się na „szpulkę”, wybieramy *Create Constant* i wpisujemy wartość 10. Połączmy teraz ikonę „i” z krawędzią pę-

tli. Na krawędzi powstanie tzw. tunel wyjściowy przekazujący dane z pętli na zewnątrz. Po zewnętrznej stronie pętli należy korzystając z *Create Indicator* wygenerować wskaźnik. Tak samo wewnątrz pętli na połączeniu należy wygenerować wskaźnik można nazwać go wewnętrzny. Jeszcze raz połączmy ikonę z krawędzią pętli. Tym razem kliknijmy prawym klawiszem myszki na powstałym tunelu wyjściowym i wybierzmy *Tunnel Mode -> Last Value* (w starszych wersjach LabView – *Disable Indexing*). Na zewnątrz pętli przy tym tunelu kreujemy wskaźnik i nazywamy go „zewnętrzny". Diagram powinien wyglądać jak na **rysunku 11**.

Proponuję umieszczenie obok siebie okien, włączenie animacji (przycisk *Highlighting Execution*) i uruchomienie programu. Oczywiście, pętla wykona się 10 razy, a wskaźnik nazwany „wewnętrzny" będzie wyświetlał numery kolejnych iteracji od 0 do 9. Jego wartość będzie aktualizowana przy każdym obiegu pętli. Wskaźniki umieszczone na zewnątrz pętli wyświetlą wartości dopiero po 10 iteracjach, wskaźnik zewnętrzny wskazuje numer ostatniej iteracji, natomiast *Array* jest tablicą zawierającą numery wszystkich iteracji.

Skąd wzięła się tablica? Otóż jest to jedna z cech pętli umożliwiająca automatyczne tworzenie tablic. Po każdej iteracji dane gromadzone są w tablicy – mogą z niej być również pobierane.

Wspomniałem wcześniej, że nie zawsze musimy określać liczbę iteracji pętli. Kiedy pobieramy dane z tablicy pętla wykona się tyle razy, ile elementów się w niej znajduje. Sprawdźmy to, tworząc nowy czysty diagram Vi. Umieśćmy na diagramie pętlę *for* – proponuję z poprzedniego programu skopiować tablicę *Array*. Następnie, w oknie diagramu, klikając na niej prawym przyciskiem myszy należy wybrać *Change to Constant*. Uzyskamy w ten sposób stałą tablicową, do której proponuję wpisać kilka przykładowych liczb. Podłączamy ją do krawędzi pętli. Zwróćmy uwagę, że w tym momencie przycisk *Run* nie jest już przełamany, więc możemy uruchomić nasz program bez określania liczby iteracji.

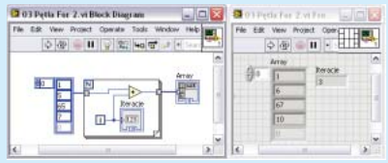
Dodajmy jeszcze kilka elementów. Diagram powinien wyglądać jak na **rysunku 12**. Będzie on dodawał do elementów z tablicy po lewej stronie numer iteracji i zapisywał do wskaźnika „wyniki". Wskaźnik „Iteracje" zawiera numer ostatniej iteracji (pamiętajmy, że jest to N-1).

Rejestry przesuwne

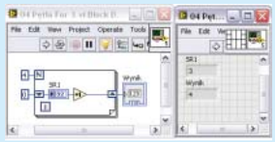
Pętle w LabView, oprócz gromadzenia elementów w tablicach, mają jeszcze inne właściwości niedostępne w typowych językach programowania. Można za ich pomocą tworzyć rejestry przesuwne. Określenie to bardziej jest znane elektronikom niż programistom. Zadaniem rejestrów jest przekazywanie informacji pomiędzy poszczególnymi iteracjami pętli.

Otwórzmy nowy dokument i umieśćmy w nim pętlę *for*. Skonfigurujmy ją w taki sposób, aby wykonała się 4 razy. Aby umieścić rejestr przesuwny klikamy na krawędzi pętli prawym przyciskiem myszy i z menu wybieramy *Add Shift Register*. Z lewej strony do rejestru przesuwnego dołączamy stałą i wpisujemy 0. Wewnątrz umieszczamy funkcję inkrementacji i wskaźnik wyświetlający dane na panelu czołowym. Gotowy diagram pokazano na **rysunku 13**.

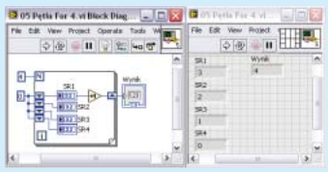
Proponuję uruchomienie programu w trybie animacji i prześledzenie jego działania. Przy pierwszej iteracji do rejestru przesuwnego zostanie przekazana wartość początkowa, która wewnątrz pętli jest pobierana od lewej strony. Następnie, po zwiększeniu o 1, jest zapisywana do rejestru przesuwnego z prawej strony. Przy kolejnej iteracji poprzednio zapisana wartość jest dostępna w rejestrze przesuwnym z lewej strony itd. Pokazany tutaj przykład inicjalizuje rejestr przesuwny wartością 0. Można pominąć inicjalizację (w pewnych przypadkach jest to wskazane), ale musi to być zrobione świadomie, ponieważ przy kolejnych uruchomieniach (wywołaniach) programu rejestr będzie przechowywał wartości z poprzedniego wywołania, co może skutkować niespodziewanymi rezultatami działania. Dlatego zawsze, jeśli jest to możliwe, należy inicjalizować rejestr wartością początkową.



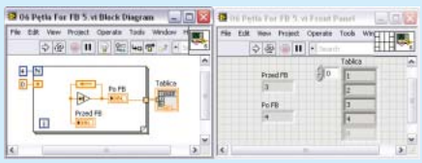
Rysunek 12. Pętla *for* – pobieranie danych z tablicy



Rysunek 13. Pętla *for* – Rejestr przesuwny



Rysunek 14. Rejestr przesuwny – przykład 2



Rysunek 15. Sprzężenie zwrotne

Zmodyfikujmy teraz program, aby wyglądał jak na **rysunku 14**. W tym celu klikamy na rejestrze przesuwnym prawym klawiszem myszy i wybieramy *Add Element* lub rozciągamy go tak, aby zawierał 4 elementy. Umieśćmy w środku pętli wskaźniki. Na panelu czołowym proponuję ułożenie elementów, tak jak na **rysunku 14**. Wskaźnik SR1 odpowiada elementowi 1 SR2 elementowi 2 itd.

Po uruchomieniu programu widać jak przesuwają się wartości w rejestrze. Zmniejszają się przy każdej iteracji o 1. W ten sposób można uzyskać informację nie tylko o danych z poprzedniej iteracji, ale również wcześniejszych.

Feedback Node

Podobne działanie do rejestrów przesuwnych ma *Feedback Node*, czyli sprzężenie zwrotne. Dzięki niemu możemy również uzyskiwać informacje o wyniku ostatniej iteracji, ale diagram jest znacznie bardziej przejrzysty i zrozumiały. Jeśli potrzebujemy danych z jeszcze wcześniejszych iteracji, to jest korzystniej stosować rejestry przesuwne. Struktura ta jest dostępna w palecie struktur. Aby umieścić ją w pętli należy przeciągnąć z palety do wnętrza pętli.

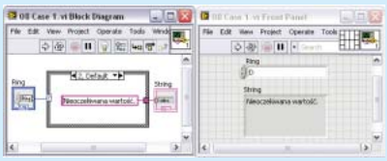
Przygotujmy program jak na **rysunku 15** i prześledźmy jego działanie. Po umieszczeniu struktury *Feedback Node* wewnątrz pętli wygląda ona inaczej niż na rysunku. Terminal do inicjalizacji znajduje się pod strzałką. Można to zmienić klikając prawym przyciskiem myszy i wybierając *Move Initialize One Loop Out*. Teraz terminal, do którego przekazujemy wartość początkową znajduje się z lewej strony pętli. Podobnie jak w dla rejestru przesuwnego polecam dbałość o inicjowanie zawartości, ponieważ obowiązują tutaj takie same zasady, jak dla rejestrów przesuwnych.

Rysunek 16. While jako główna pętla programu

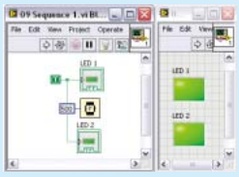
Pętla while

Kolejną pętlą dostępną w LabView jest *while*. Diagram zawarty w pętli wykonuje się dopóki, dopóty warunek kontynuacji jest prawdziwy. Sprawdzenie warunku odbywa się na końcu pętli, a więc wykona się ona przynajmniej raz. Wewnątrz pętli znajduje się licznik „i" zawierający numer aktualnej iteracji. Z prawej strony znajdziemy ikonę kontynuacji. Przekazana do niej wartość logiczna decyduje o kontynuacji pętli. Klikając prawym przyciskiem myszki na ikonie możemy określić warunek kontynuacji:

- *Stop if True* – zatrzymaj, jeśli warunek logiczny jest 1 (prawdziwy).
- *Continue if True* – kontynuuj, jeśli warunek logiczny jest 1.



Rysunek 17. Przykład użycia struktury *Case*



Rysunek 18. Zaświecenie diod LED bez użycia struktury sekwencji

Pętla ta jest stosowana w przypadkach, gdy nie możemy określić liczby iteracji, na przykład jako pętla główna programu (**rysunek 16**).

Wszystkie elementy związane z pętlą *for*, czyli: rejestry przesuwne (*Shift Register*), sprzężenie zwrotne *(Feedback Node)* oraz indeksowanie, dotyczą również pętli *while*. Z tą tylko małą różnicą, że domyślnie tunel wyjściowy jest skonfigurowany jako *Last Value* (*Disable Indexing* w starszych wersjach LabView), czyli otrzymujemy liczbę, a nie tablicę. Oczywiście, indeksowanie można włączyć.

Struktura case

Kolejnym elementem języka G jest struktura *Case*. Funkcjonalnie odpowiada ona instrukcji *switch* znanej z języka C. Na podstawie wartości zmiennej umożliwia wybranie i wykonanie właściwego fragmentu programu. Akceptuje zmienne logiczne, numeryczne i tekstowe oraz specjalną zmienną niosącą informacje o błędzie. Ma postać nakładających się na siebie ramek. W każdej z nich umieszcza się kod odpowiadający danemu przypadkowi.

Spróbujmy sprawdzić działanie struktury. W nowym, czystym dokumencie na panelu czołowym z palety *Ring & Enum* wstawiamy *Text Ring*. Klikając prawym przyciskiem myszki wybieramy *Edit Items*. Teraz klikamy na przycisk *Insert* i w podświetlonym polu wpisujemy np. literkę „D". Następnie ponownie naciskamy *Insert* i wpisujemy literkę „C", ponownie – „B" i ponownie – „A". Zamykamy okno przyciskiem OK. Nasz *Text Ring* zawiera teraz cztery pola wyboru opisane za pomocą liter A, B, C i D. Po wybraniu opcji A, funkcja zwróci wartość 0, dla opcji B wartość 1 itd. Przejdźmy teraz na diagram i z palety *Structures* wstawiamy *Case Structure* (tak samo, jak pętlę). Podłączamy nasz *Text Ring* do terminala ze znakiem pytajnika na lewej krawędzi pętli. Struktura zawiera teraz dwie ramki, opisane jako „0" i „1". Pomiędzy ramkami można przechodzić za pomocą górnej krawędzi pętli klikając w małe strzałeczki lub rozwijając listę wyboru. Do ramki „0" wstawmy z palety *String -> String Constant* i wpiszmy tekst „Wybrano opcję A". Następnie podłączmy stałą do krawędzi pętli. Pojawi się tunel wyjściowy z białym kwadratem w środku. Zwróćmy uwagę, że przycisk *Run* jest przełamany sygnalizując błąd. Przyczyną błędu jest nieokreślenie wartości zwracanej przez tunel wyjściowy w przypadku, gdy wykona się kod zawarty w ramce 1. Przejdźmy do ramki 1 i podłączmy do tego samego tunelu stałą tekstową, wpiszmy „Wybrano opcję B". Teraz już nie ma informacji o błędzie. Dla każdego przypadku jest niezbędne określenie wartości tuneli wyjściowych.

Tunele wejściowe mogą przekazywać wartości tylko do jednej ramki. Klikając prawym przyciskiem myszy na krawędzi ramki wybierzmy *Add Case After*. Spowoduje to dodanie ramki z numerem 2 – tam również wstawmy stałą podłączoną do tunelu wyjściowego i wpiszmy „Nieoczekiwana wartość". Znów klikamy na krawędzi pętli, z menu wybieramy *Make This The Default Case*. Obok liczby 2 pojawi się napis *Default*. Oznacza to, że dla niezdefiniowanych przypadków wykona się kod zawarty w tej ramce. Tu będzie to dla opcji „C" i „D". Nie wymaga się określania, która z ramek jest określona jako *Default* tylko w wypadku zmiennych logicznych. Na **rysunku 17** pokazano przykładowy program, wyświetlający w polu *string* napis zależny od wybranej wartości w kontrolce *ring*.

Struktura sequence

Struktura sekwencji umożliwia wykonanie instrukcji w odpowiedniej kolejności. Jest szczególnie przydatna w miejscach, w których fragmenty kodu nie są ze sobą bezpośrednio połączone, to znaczy – nie przekazują sobie żadnych parametrów. Takie funkcje będą wykonywane pseudorównolegle. Nie jesteśmy w stanie określić, która wykona się pierwsza. Z pomocą przychodzi nam struktura *Sequence*. W LabView została zaimplementowana w dwóch postaciach. *Flat Sequence Structure* charakteryzuje się tym, że na diagramie widoczne są jednocześnie wszystkie ramki i funkcje w nich zawarte. Daje to przejrzystość kodu i umożliwia łatwą analizie. Dane pomiędzy ramkami przekazywane są w łatwy sposób przez tunele. Druga postać to *Stacked Sequence Structure*. Jest w niej tylko jedna ramka, a pozostałe są ukryte pod spodem. Przekazywanie wartości pomiędzy ramkami odbywa się z wykorzystaniem specjalnego zacisku. Korzystanie z niej jest korzystne w wypadku, gdy kod zawarty w ramce jest duży i ramki nie mieszczą się na monitorze.

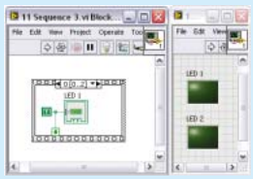
Strukturę sekwencji często wykorzystuje się podczas komunikacji z przyrządem pomiarowym, gdzie po wysłaniu polecenia musimy odczekać pewien czas, aby urządzenie mogło np. wykonać pomiar i odesłać wynik. Ponieważ nie wszyscy dysponują przyrządem pomiarowym i nie umiemy ich jeszcze programować, przygotujemy prosty przykład polegający na zaświeceniu dwóch diod LED w odpowiedniej kolejności: zaświecenie się LED 1 symbolizuje wysłanie polecenia do przyrządu, a LED 2 otrzymanie odpowiedzi.

Program powinien:

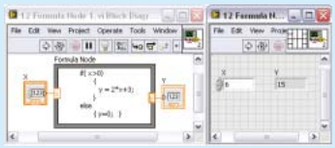
- zapalić diodę LED 1
- odczekać 0.5s
- zapalić diodę LED 2

Otwórzmy teraz nowy Vi. W oknie diagramu umieszczamy dwie diody z palety *Boolean*. Na diagramie z palety *Timing* pobieramy funkcję *Wait*, podpinamy do niej stałą i wpisujemy 500 – spowoduje ona zwłokę 500 ms. Do diod podpinamy stałą o wartości 1. Diagram powinien wyglądać jak na **rysunku 18**.

Rysunek 19. Przykład użycia *Flat Sequence*



Rysunek 20. Przykład użycia *Stacked Sequence*



Rysunek 21. Przykład użycia *Formula Node*

Dociekliwi mogą sprawdzić, że na kolejność wykonywania programu nie wpływa umieszczenie ikon na diagramie ani kolejność ich wstawiania. To środowisko decyduje, w jakiej kolejności zostaną wykonane wszystkie procedury. Uruchamiając program zauważymy jednoczesne, natychmiastowe zaświecenie się diod i półsekundowe opóźnienie przed zakończeniem programu. Dzieje się tak, ponieważ funkcje umieszczone na diagramie są wykonywane równolegle, jak niezależne programy. Aby uzyskać pożądaną kolejność wykonywania instrukcji należy użyć sekwencji. Korzystając ze struktury *Flat Sequence Structure*, dostępnej w palecie struktur, zmodyfikujmy nasz diagram, aby wyglądał jak na **rysunku 19**.

Po wstawieniu struktury klikamy prawym przyciskiem myszki i wybieramy *Add Frame After* (dodaje ramkę po aktualnej) lub *Add Frame Before* (dodaje ramkę przed aktualną). Teraz wszystko wykona się we właściwej kolejności. Diagram jest przejrzysty a przekazywanie danych jest zrozumiałe.

Zróbmy taki sam program korzystając ze struktury *Stacked Sequence Structure*. Program będzie działał dokładnie tak samo, a różnice polegają na wyglądzie struktury oraz przekazywaniu parametrów pomiędzy ramkami. Aby przesłać wartość ze stałej znajdującej się w pierwszej ramce do kolejnych, musimy kliknąć prawym przyciskiem myszki na krawędzi ramki z menu lokalnego i wybrać *Add Sequence Local*. Spowoduje to wstawienie na krawędzi ramki zacisku, do którego dołączamy przekazywany parametr. Po tej operacji, w terminalu zostanie wyświetlona strzałka skierowana na zewnątrz ramki, oznaczająca miejsce przekazania wartości do zacisku. Od tej pory w kolejnych ramkach znajduje się zacisk ze strzałką skierowaną do ośrodka, z którego w dowolnej ramce możemy pobierać przekazywany parametr. Struktura ta jest wykorzystywana w przypadku dużej liczby sekwencji, aby ograniczyć wielkość diagramu. Przykład użycia *Stacked Sequence* pokazano na **rysunku 20**.

Formula Node

Formula Node jest strukturą przydatną podczas wykonywania obliczeń matematycznych. Umożliwia zapisanie kodu w postaci podobnej do tradycyjnych języków programowania i pozwala na uniknięcie tworzenia dużych diagramów w wypadku skomplikowanych obliczeń.

Składnia jest podobna do języka C, dostępne są funkcje matematyczne, takie jak sinus, cosinus i inne oraz instrukcje warunkowe np. *if*, pętle *Do*, *While*, *For* i inne elementy dostępne w języku C. Linie kodu mogą zawierać komentarze jednoliniowe poprzedzone „//” lub wieloliniowe umieszczone pomiędzy znakami „/*” (początek bloku tekstu) i „*/” (koniec bloku tekstu).

W nowym Vi umieszczamy z palety *Structures -> Formula Node*. Aby do „wnętrza” przekazać parametry musimy dodać zaciski wejściowe. W tym celu klikamy na krawędzi pętli i wybieramy *Add Input*. W powstałym kwadracie wpisujemy nazwę zmiennej – będziemy się nią posługiwali wewnątrz struktury. Analogicznie poprzez *Add Output* dodajemy wyjścia zwracające wynik obliczeń. Na **rysunku 21** pokazano przykład użycia tej struktury.

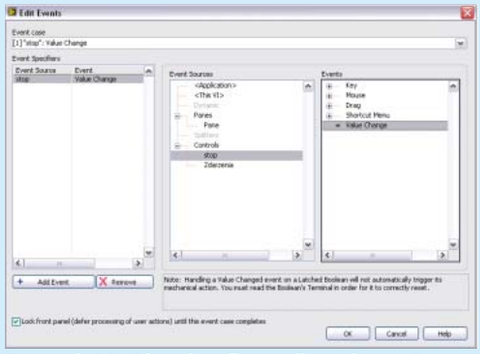
Struktura zdarzeń – Event

Struktura *Event* jest bardzo przydatna podczas budowania interfejsu użytkownika. Reaguje na zdarzenia występujące na panelu czołowym, np. najechanie myszką na wybrany element, kliknięcie w niego, podwójne kliknięcie, zmianę wartości i inne. Najczęściej jest umieszczana w pętli głównej programu, ponieważ poza nią wykonałaby się tylko raz, reagując tylko na zdarzenie, które wystąpiło pierwsze. Aby poznać jej działanie, przygotujemy program reagujący na kilka zdarzeń. Informacja o rodzaju zdarzenia będzie wyświetlana w polu tekstowym.

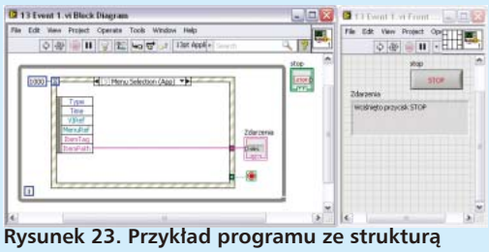
W nowym Vi wstawiamy strukturę *Event*. Domyślnym, zdefiniowanym zdarzeniem jest *Timeout*. W prawym, górnym rogu znajduje się terminal *Event Timeout*. Klikając na nim prawym przyciskiem myszki wygenerujmy *Create Constant*. Pojawi się stała podłączona do terminalu z domyślną wartością -1. W praktyce oznacza to, że struktura będzie czekała na wystąpienie któregoś ze zdefiniowanych zdarzeń przez nieskończenie długi czas, więc po uruchomieniu działanie naszego programu nigdy nie zakończy się. Zmieniamy stałą -1 na 1000, co zapobiegnie tej sytuacji i spowoduje, że struktura będzie oczekiwała przez 1 s (1000 ms) na wystąpienie zdarzeń innych niż *Timeout*, a po upływie tego czasu wykona się kod zawarty w ramce *timeout*. Nie ma zdefiniowanych żadnych zdarzeń, więc po upływie 1 s program zakończy się.

Na panelu czołowym umieśćmy dwa elementy: przycisk *Stop* oraz *String Indicator* (nazwijmy go *Zdarzenia*). Wskaźnik *zdarzenia* umieśćmy poza strukturą *Event*. W ramce *timeout* umieśćmy stałą tekstową z komunikatem „Wystąpiło zdarzenie Timeout” i podłączmy ją do kontrolki *Zdarzenia*. Po uruchomieniu programu i upływie 1 s wystąpi zdarzenie *Timeout* i zostanie wyświetlony tekst zawarty w ramce *timeout*. Obejmijmy teraz pętlą *While* strukturę i dodajmy zdarzenie związane z przyciskiem *Stop*. W tym celu klikamy prawym przyciskiem myszy na krawędzi struktury *Event* i z menu wybieramy *Add Event Case…*. Otworzy się okno edycji jak na **rysunku 22**.

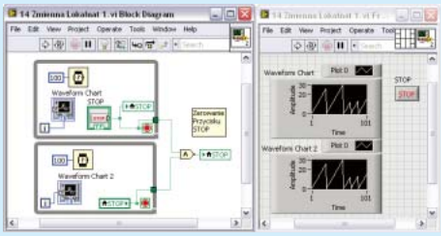
W oknie *Event Specifiers* znajdują się zdarzenia zdefiniowane dla wybranej ramki, a w *Event So-*



Rysunek 22. Okno konfiguracji struktury *Event*



Rysunek 23. Przykład programu ze strukturą *Event*



Rysunek 24. Przykład zastosowania zmiennej lokalnej

urces – dostępne źródła zdarzeń. Na diagramie można definiować zdarzenia związane z aplikacją, danym programem, oknem programu i kontrolkami. W oknie *Event* są widoczne zdarzenia możliwe do zdefiniowania dla wybranego elementu. Dla przycisku *Stop* konfigurujemy zdarzenie *Value Change* (zmiana wartości). Po tym fakcie, każda zmiana wartości spowodowana przez użytkownika wygeneruje zdarzenie. Należy pamiętać, że struktura reaguje tylko na zdarzenia wywołane przez użytkownika, więc zmiana wartości w sposób programowy nie spowoduje wygenerowania zdarzenia. Po skonfigurowaniu zamykamy okno. Pojawi się nowa ramka, w której należy umieścić stałą logiczną o wartości *true* i podłączyć ją do zacisku kontynuacji pętli. Wewnątrz ramki umieszczamy jeszcze stałą tekstową z komunikatem „Wciśnięto przycisk Stop" i łączymy ją z tunelem wyjściowym prowadzącym do wskaźnika *Zdarzenia*. Teraz można przetestować pracę programu. Zwróćmy uwagę na to, że przycisk *Stop* nie jest nigdzie dołączony, nie musi znajdować się w pętli ani strukturze, może być umieszczony w dowolnym miejscu na diagramie. Zmiana jego wartości jest identyfikowana za pomocą struktury *Event*. Popatrzmy jeszcze na tunel wyjściowy prowadzący do zacisku kontynuacji pętli. Znajduje się w nim kropka sygnalizująca, że nie zdefiniowano wartości zwracanej przez ten tunel dla każdej ramki (każdego zdarzenia). Jeśli klikniemy prawym przyciskiem myszki na tunelu, to zobaczymy, że jest zaznaczona opcja *Use Default If Unwired*. Możemy ją pozostawić bez zmian, ponieważ domyślnie pętla wykonuje się dopóty, dopóki warunek kontynuacji jest równy *false*. Należy jednak pamiętać o tym, aby dla każdego zdarzenia wartości zwracane przez tunele wyjściowe były prawidłowe.

Aby zobaczyć zalety stosowania struktury, dodajmy jeszcze dwa zdarzenia. Jak poprzednio dodajemy nową ramkę zdarzeń w oknie *Event Sources*, wybieramy *Controls -> Stop*, a w oknie *Events ->Mouse -> Mouse Move* wstawmy w nowoutworzonej ramce komunikat „Myszka przesunęła się nad przyciskiem Stop". Ramkę dołączamy do odpowiedniego tunelu wyjściowego. Kod znajdujący się w tej ramce wykona się, gdy kursor myszy przemieści się nad przyciskiem *Stop*.

Dodajmy jeszcze jedno zdarzenie. Po otwarciu okna konfiguracji, w oknie *Event Sources* wybieramy *<This VI>*, a w oknie *Events -> Menu Selection (App)*. Z lewej strony nowopowstałej ramki, pojawi się kilka terminali, z których można pobrać przydatne informacje dotyczące zdarzenia. Połączmy terminal *ItemPath* z naszym tunelem wyjściowym. Kod znajdujący się w tej ramce wykona się po wybraniu dowolnej, aktywnej opcji z górnego menu aplikacji, a w wskaźniku *Zdarzenia* znajdzie się informacja o wybranej opcji menu. Na **rysunku 23** pokazano program z ramką obsługującą *Menu Aplikacji*.

Kilka słów o terminalach znajdujących się z lewej strony, z których możemy pobierać przydatne informacje. Dla każdego typu zdarzeń dostępne są wspólne terminale, takie jak rodzaj zdarzenia (*Type*) i czas jego wystąpienia (*Time*) oraz specyficzne dla danego zdarzenia, jak na przykład: poprzednia wartość (*OldVal*), nowa wartość (*NewVal*), zmiana wartości (*Value Change*).

Obsługa zdarzeń pozwala na łatwe i szybkie przygotowanie przejrzystego interfejsu użytkownika. Można przy tym korzystać ze zdarzeń dotyczących aplikacji lub poszczególnych elementów panelu czołowego. Dzięki temu możemy zbudować aplikację obsługującą przenoszenie elementów za pomocą myszki.

Zmienne lokalne

Zmienna lokalna jest dostępna tylko w obrębie jednego diagramu. Pozwala na zapis lub odczyt wartości w różnych jego miejscach bez prowadzenia połączeń. Zmienną identyfikuje się po nazwie elementu na panelu czołowym. Może być przypisana zarówno do kontrolki (*controls*) jak i do wskaźnika (*indicator*). Zapis wartości do zmiennej związanej z kontrolką pozwala na jej użycie jako wskaźnika – jest możliwa również sytuacja odwrotna.

Zmienną lokalną można utworzyć na dwa sposoby. Pierwszy polega na wstawieniu w oknie diagramu symbolu zmiennej lokalnej z palety *Structures* po kliknięciu myszką i wskazaniu na liście zmiennej, którą ma reprezentować. Drugi polega na kliknięciu prawym klawiszem myszy na ikonie elementu z panelu czołowego i wybraniu opcji *Create -> Local Variable*. Wtedy zostanie wygenerowana zmienna lokalna reprezentująca dany element. Klikając na symbolu zmiennej prawym przyciskiem myszki można wybrać czy dane mają być do niej zapisane *(Change To Write)*, czy z niej odczytywane (*Change To Read)*. Gdy ramka symbolu zmiennej jest pogrubiona i terminal znajduje się z prawej strony, to jest skonfigurowana w taki sposób, aby odczytywać z niej dane. Gdy jest cienka, a terminal z lewej strony, to można zapisywać do niej dane.

Przygotujmy teraz przykład, w którym za pomocą zmiennej lokalnej będziemy sterowali dwiema pętlami. W nowym Vi umieśćmy obok siebie dwie pętle *While*. Dodajmy przycisk *Stop*, podłączmy go do zacisku kontynuacji (czerwona kropka) jednej z pętli. Korzystając z opisu powyżej umieścimy zmienną lokalną reprezentującą przycisk *Stop* i podłączmy ją do zacisku kontynuacji drugiej pętli. W obu pętlach wstawmy funkcję opóźniającą *Wait* oraz wskaźnik *Waveform Chart*. Następnie trzeba go dołączyć do licznika iteracji „i". Gotowy program powinien wyglądać jak na **rysunku 24**.

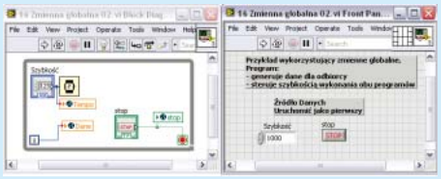
Ponieważ pojawił się błąd związany z niewłaściwą akcją przycisku *Stop*, należy – klikając prawym przyciskiem myszki na przycisku w oknie panelu czołowego – wybrać opcję *Mechanical Action -> Switch*

Rysunek 25. Wygląd panelu czołowego zmiennej globalnej.

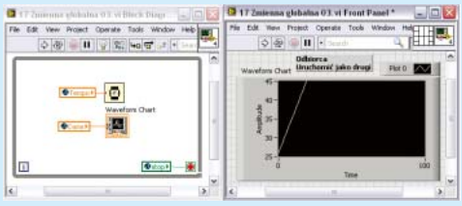
When Pressed. Wspomnę przy tym, że umieszczone obok siebie pętle wykonują się pseudorównocześnie.

Zmienne globalne (Global Variable)

Zmienna globalna jest szczególnym przypadkiem przyrządu wirtualnego – ma panel czołowy, ale nie ma diagramu. Jest dostępna nie tylko w oknie aktualnego diagramu, ale może przekazywać dane innym programom (instrumentom wirtualnym).



Rysunek 26. Przykład zastosowania zmiennej globalnej – program zapisujący dane do zmiennej



Rysunek 27. Przykład zastosowania zmiennej globalnej – program odbierający dane

Aby utworzyć zmienną globalną należy z palety Structures wstawić jej symbol w oknie diagramu. Następnie, poprzez dwukrotne kliknięcie na nim, otworzyć okno podobne do panelu czołowego. Można w nim umieścić kilka zmiennych różnego typu i nadać im odpowiednie nazwy – posłużą one do ich identyfikacji.

Zmienną globalną można przygotować również w inny sposób poprzez wybranie z menu górnego *File -> New…* Następnie, w otwartym oknie, wybieramy *Other Files -> Global Variable*. W ten sposób otwieramy okno, w którym umieszczamy zmienne. Gotowy plik należy zapisać na dysku.

W przykładzie demonstrującym działanie zmiennych globalnych użyjemy dwóch programów: jeden będzie źródłem danych, drugi przedstawi na wykresie odebrane dane. Zaczynamy od przygotowania zmiennej globalnej. W nowym Vi wstawiamy symbol zmiennej globalnej, otwieramy jej panel czołowy i umieszczamy trzy zmienne przyciski *Stop* oraz dwie kontrolki numeryczne *Dane* i *Tempo*. Panel zmiennej powinien wyglądać jak na **rysunku 25**. Teraz należy zapisać zmienną wybierając z menu *File ->Save* nadając jej nazwę np. *Zmienna Globalna*.

Wracamy do diagramu. Wstawiamy pętle *While* oraz funkcję *Wait*. Na panelu czołowym dodajemy przycisk *Stop* i kontrolkę numeryczną, którą nazywamy np. *Szybkość*. Posłuży ona do ustawiania opóźnienia pomiędzy kolejnymi iteracjami pętli. Wracamy do diagramu i dodajemy zmienne globalne – mając pierwszą zmienną możemy ją skopiować. Można też, klikając prawym przyciskiem myszy w oknie diagramu po otworzeniu się okna z paletami funkcji, wybrać znajdującą się na samym dole opcję *Select a VI…* i wskazać miejsce zapisania zmiennej na dysku. Będą potrzebne 3 zmienne. Gdy już są, to po najechaniu na zmienną i pojawieniu się „łapki" klikamy i wybieramy właściwą nazwę. Łączymy wszystko, jak na **rysunku 26**. Gotowy program należy zapisać na dysku.

Pozostało jeszcze przygotowanie programu odczytującego dane ze zmiennych globalnych. Otwieramy znów nowy dokument i na panelu czołowym umieszczamy wykres *Waveform Chart* z palety *Graph*. Przechodzimy na diagram i umieszczamy pętlę *While*, funkcję *Wait* oraz trzy zmienne globalne za pomocą opcji *Select a VI…* Klikając prawym przyciskiem myszy na zmiennych wybieramy *Change To Read* i łączymy wszystko, jak na **rysunku 27**.

Mamy już wszystko, co potrzebne. Jako pierwszy należy uruchomić program będący źródłem danych w kontrolce *Szybkość* – proponuję wpisanie np. 1000, aby nasz program nie działał zbyt szybko. Przy każdej iteracji pętli do zmiennej *Dane* zapisany zostanie numer iteracji. Podobnie z pozostałymi zmiennymi. Drugi program przy każdej iteracji odczytuje dane ze zmiennej globalnej *Dane* i przedstawia je na wykresie. Są to numery iteracji z programu zapisującego dane. Poprzez zmienną *Tempo* możemy sterować szybkością wykonania pętli, a dzięki zmiennej *Stop* zdalnie zakończyć pracę programu.

Podsumowanie

Do tej pory poznaliśmy środowisko oraz podstawowe funkcje strukturalne języka G. W kolejnej części zajmiemy się operacjami na tablicach i klastrach danych.

Wiesław Szaj
wszaj@prz.edu.pl