Podstawy projektowania aplikacji w LabWindows/CVI

(opr. Bogdan Kasprzak)

1. Wprowadzenie.

Systemy pomiarowe realizuje się często w postaci tzw. przyrządów wirtualnych. Przyrząd wirtualny bazuje na określonym systemie komputerowym oraz zestawie wybranych urządzeń pomiarowych stanowiących urządzenia I/O systemu komputerowego. Oprogramowanie integruje działanie całości w celu realizacji określonych zadań pomiarowych. Istotnym składnikiem programu aplikacyjnego jest graficzny interfejs użytkownika (GUI) zapewniający interakcyjną współpracę z użytkownikiem. W formie wizualnej stanowi on programową realizację panelu czołowego przyrządu zawierającego elementy nastawcze (pokrętła, przełączniki, przyciski itp.) oraz elementy prezentacyjne (wyświetlacze numeryczne, graficzne itp.).

Kreowanie GUI aplikacji może być trudne. Jednocześnie projektanci systemów pomiarowych nie są w większości przypadków profesjonalnymi programistami. Trudności programistyczne dotyczą również szeregu innych aspektów projektowanej aplikacji, np. sterowania urządzeniami I/O, akwizycji danych itd. Często aplikacja musi uwzględnić specyficzne wymagania dotyczące szybkości i wydajności działania, które mogą być spełnione tylko przez użycie języka programowania takiego jak C, co wymaga od projektanta doskonałej jego znajomości. Są więc potrzebne narzędzia ułatwiające kreowanie aplikacji pomiarowych w języku C dla najczęściej wykorzystywanych systemów operacyjnych (przede wszystkim Windows). Jednym z przykładów takiego narzędzia jest środowisko projektowe LabWindows/CVI.

2. System operacyjny Windows.

Windows jest wielozadaniowym systemem operacyjnym z graficznym interfejsem użytkownika. Wiele zadań może być wykonywanych jednocześnie a wszystkie akcje użytkownika są odbierane poprzez okna.

Proces i wątek. Uruchomienie aplikacji w systemie Windows oznacza inicjalizację procesu (*process*), który kreuje co najmniej jeden wątek (*thread*), ładuje kod programu oraz wymagane biblioteki dynamiczne (DLL). Wątek reprezentuje jedno z zadań potrzebnych do wykonania całej pracy. Proces może być jednowątkowy lub wielowątkowy. Proces wielowątkowy jest użyteczny w sytuacjach, gdy pewne zadanie wymaga znacznego czasu. Takie zadanie może być realizowane przez jeden wątek, podczas gdy inne zadania tego procesu są realizowane przez oddzielne wątki. Inną korzyścią jest fakt, że w systemach wieloprocesorowych (np. Windows NT) wątki mogą być wykonywane współbieżnie, każdy przez oddzielne procesory.

Wielozadaniowość i szeregowanie. Windows jest środowiskiem wielozadaniowym, w którym wiele programów jest wykonywanych równocześnie. Procesor może w danym odcinku czasu wykonywać jeden wątek. Wielozadaniowość w Windows uzyskuje się przez okresowy przydział krótkich odcinków czasu procesora poszczególnym wątkom należącym do jednego lub różnych procesów. W rezultacie przetwarzanie jest przełączane tam i z powrotem pomiędzy wątkami.

W systemie Windows zastosowano wielozadaniowość z wywłaszczaniem, która zapobiega monopolizacji procesora przez jedną aplikację. Zamiast oczekiwania aż wątek dobrowolnie zwolni procesor, system operacyjny przerywa przetwarzanie wątku po upływie przydzielonego przedziału czasu lub po odbiorze żądania z wątku mającego wyższy priorytet. Taki sposób zapewnia wszystkim wątkom potrzebny dostęp do procesora. Ogólne zasady uzyskania wielozadaniowości są następujące:

- Przetwarzanie wątku do momentu przerwania lub do momentu, gdy wątek musi czekać na dostępność zasobu.
- Zapamiętanie kontekstu wykonywania wątku.
- Załadowanie kontekstu kolejnego wątku.
- Powtarzanie powyższej sekwencji dopóki istnieje wątek oczekujący na wykonanie.

Kluczowym problemem wielozadaniowości jest proces szeregowania wątków (*scheduling*), czyli określenia, który z nich powinien być wykonywany jako następny. Podczas kreacji wątku, przyjmuje on zadany priorytet wykonywania lub jest mu on przydzielany. Dodatkowo wątki mają różne stany, takie jak nieaktywny przez określony okres czasu, oczekujący na pewne zdarzenie lub gotowy

do wykonania. Ogólnie wątek o najwyższym priorytecie będący z stanie gotowości do wykonania jest brany jako kolejny do realizacji. Jest on wtedy ustawiany w stan realizacji, w którym pozostaje do momentu wywłaszczenia lub wejścia w stan oczekiwania na określone zdarzenie.

Komunikaty i okna. Okna są wizualnymi elementami charakteryzującymi aplikacje systemu Windows. Są one podstawową metodą komunikacji aplikacji z użytkownikiem. Podobnie użytkownik używa okno do komunikacji z aplikacją w celu osiągnięcia określonego postępowania podczas realizacji zadania. Środowisko Windows jest systemem bazującym na komunikatach. Każde zdarzenie w systemie takie jak przesunięcie myszki lub wciśnięcie klawisza klawiatury powoduje wygenerowanie komunikatu. Komunikat zawiera informacje o typie zdarzenia, czasie jego wystąpienia oraz oknie, do którego jest on skierowany.

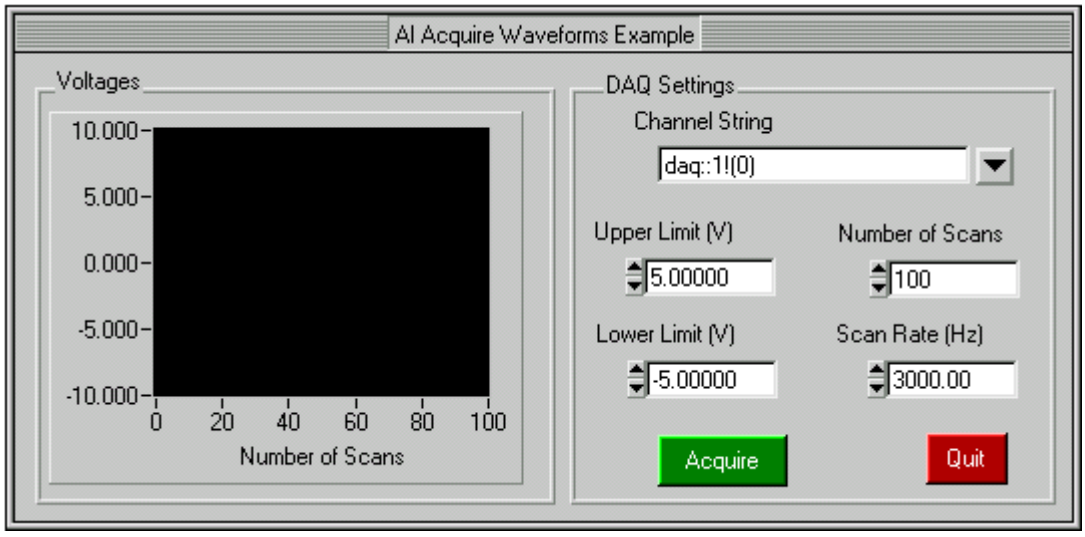
Aplikacja systemu Windows rozpoczyna swoje działanie od funkcji o nazwie WinMain. Funkcja ta kreuje jedno lub więcej okien. Każde okno zawiera z kolei procedurę (WndProc), która jest odpowiedzialna za określenie co okno wyświetla i w jaki sposób okno odpowiada na akcje użytkownika (komunikaty). Fragment kodu funkcji WinMain nazywany pętlą komunikatów otrzymuje komunikaty z kolejki komunikatów i zwraca je do Windows w celu wysłania go do właściwej procedury okienkowej (WndProc). Daje to aplikacji szanse dodatkowego przetworzenia komunikatu przed jego przekazaniem do okna.

Program sterowany zdarzeniami. Aplikacje systemu Windows działają w sposób pasywny, tzn. wykonują istotną pracę tylko w odpowiedzi na komunikaty generowane przez użytkownika, system operacyjny lub inne aplikacje. Program sterowany zdarzeniami wymaga napisania kodu, który jest zdolny do interpretacji komunikatów i odpowiedniej reakcji na dany komunikat. Zadania, które są przypisane do komunikatów obsługiwanych utworzonym kodem wymagają specyficznej akcji użytkownika w celu ich aktywizacji. Zrozumienie i przystosowanie się do paradygmatu sterowania zdarzeniami stanowi podstawę programowania aplikacji okienkowych.

3. Koncepcja budowy aplikacji w LabWindows.

Graficzny interfejs użytkownika.

Graficzny interfejs użytkownika jest wizualną częścią aplikacji pomiarowej (rys.1-1). W LabWindows jego utworzenie jest relatywnie proste dzięki specjalnemu edytorowi (*uir-editor*) stanowiącemu część pakietu LW.



Rys.1-1. Przykład graficznego interfejsu użytkownika.

Podstawowym elementem interfejsu GUI jest okno, nazywane w LW panelem. Jest on prostokątnym obszarem ekranu, który grupuje obiekty sterujące interfejsu. Obiekty sterujące umieszczone w panelu służą do przyjmowania danych od użytkownika oraz prezentacji danych. Każda aplikacja działa wewnątrz panelu. Podstawowy panel aplikacji jest nazywany macierzystym

panelem (*parent panel*). Może on posiadać panele potomne (*child panel*) a te kolejne panele potomne. Panele potomne są powiązane z panelem macierzystym w następujący sposób:

- Panel potomny jest zawsze ulokowany na szczycie panelu macierzystego.
- Panel potomny jest ograniczony przestrzenią panelu macierzystego. Część panelu potomnego wykraczająca poza tą przestrzeń nie jest kreślona.
- Główne akcje wykonywane na macierzystym panelu (ukrycie, przesunięcie, destrukcja, maksymalizacja, minimalizacja) automatycznie wpływają na panel potomny.
- System operacyjny Windows oraz LW zarządzają panelami potomnymi pośrednio przez ich panele macierzyste.

Panele potomne pozwalają na dodatkowe grupowanie obiektów sterujących interfejsu użytkownika i budowę interfejsu dynamicznie zmieniającego swoją postać wizualną.

W kodzie programu każdy panel jest charakteryzowany stałą oraz zmienną *panel handles.* Stała związana z panelem jest istotna dla działania interfejsu GUI i dlatego LW automatycznie sugeruje domyślne nazwy stałych dla paneli oraz przypisuje im wartości. W większości sytuacji propozycje nazw są wystarczające i nie muszą być zmieniane. Np. dla panelu macierzystego jest sugerowana nazwa PANEL. Zmienne typu *panelHandle* przechowują wartości numeryczne przypisane przez system Windows panelom aplikacji. Program korzystając z tych zmiennych może działać w obrębie wybranego panelu.

Wszystkie obiekty stosowane do budowy graficznego interfejsu użytkownika należą do jednej z dwóch kategorii: sterujących (*controls*) lub prezentacyjnych (*displays*). Pierwsze służą do wprowadzania różnych danych, drugie do prezentacji danych dostarczanych przez aplikację. Każdy obiekt wprowadzony do panelu jest w programie określony przez stałą *control ID.* Stała związana z obiektem jest istotna dla działania interfejsu GUI i LW automatycznie sugeruje domyślne nazwy stałych dla obiektów oraz przypisuje im wartości. W większości sytuacji propozycje nazw są wystarczające i nie muszą być zmieniane. Np. panel przełącznika binarnego wstawiany do panelu macierzystego dostanie nazwę PANEL_BINARYSWITCH (przedrostek PANEL jest nazwą stałej określającej panel macierzysty, do którego obiekt należy).

Procedura obsługi zdarzeń oraz funkcje Callback.

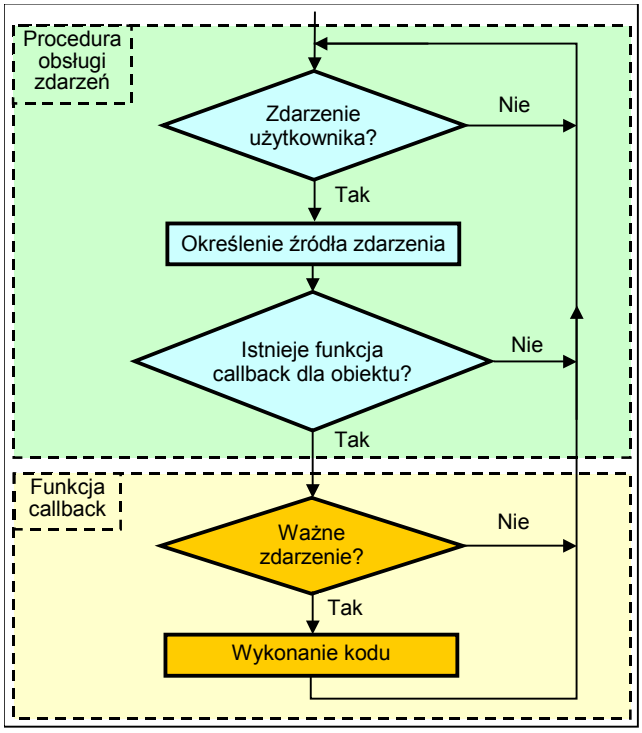
Aplikacja LW jest sterowana zdarzeniami, stąd jej rdzeniem jest procedura obsługi zdarzeń, która określa jak postąpić z wieloma napływającymi zdarzeniami użytkownika, generowanymi na skutek przesunięcia myszki lub naciśnięcia klawisza klawiatury. Procedura obsługi zdarzeń stosuje bardzo proste zasady. Najpierw określa źródło zdarzenia, czyli nazwę obiektu gdzie powstało zdarzenie. Wtedy sprawdza czy obiekt posiada procedurę realizującą operacje związane z zdarzeniami nazywaną w LW funkcją callback. Jeśli nie ma takiej funkcji, zdarzenie jest ignorowane i procedura obsługi zajmuje się kolejnym zdarzeniem lub oczekuje na nie.

Jeśli program obsługi zdarzeń znajdzie funkcję callback związaną z obiektem, przekazuje jej różne dane o zdarzeniu i inicjuje jej wykonanie, tak aby obsłużyła zaistniałe zdarzenie. Po obsłużeniu zdarzenia program przechodzi w stan oczekiwania na kolejne zdarzenie. Jak widać program obsługi zdarzeń zajmuje się wywoływaniem odpowiedniej funkcji callback i podczas jego działania sterowanie programem znajduje się w wywołanej funkcji callback, gdzie umiejscowione są wszystkie działania związane z rzeczywistą obsługą zdarzenia. Sprawne działanie aplikacji, czyli szybka obsługa zdarzeń, wymaga zatem w miarę oszczędnej konstrukcji funkcji callback.

Podsumowując, obsługę zdarzeń a tym samym funkcjonowanie programu utworzonego w LW można opisać następująco (rys.1-2):

- Program obsługi określa panel i obiekt, w którym miało miejsce zdarzenie.
- Sprawdza czy obiekt będący źródłem zdarzenia posiada zdefiniowaną funkcję callback. Jeśli funkcja taka nie istnieje, zdarzenie jest ignorowane i program obsługi wraca do oczekiwania na następne zdarzenie.
- Jeśli funkcja callback istnieje, następuje jej wywołanie z przekazaniem odpowiednich argumentów. Sterowanie programem przechodzi do wywołanej funkcji.
- Funkcja sprawdza czy przekazane zdarzenie jest na liście obsługiwanych zdarzeń, czyli czy jest ważne. Jeśli nie jest ważne zostanie zignorowane i funkcja kończy swoje działanie. Program obsługi czeka na następne zdarzenie.

- Jeśli zdarzenie jest ważne następuje wykonanie kodu przeznaczonego do jego obsługi i po jego wykonaniu powrót do programu obsługi.



Rys.1-2. Diagram obsługi zdarzenia użytkownika.

Za pomocą funkcji callback można utworzyć program zdolny do przetwarzania zdarzeń generowanych przez interfejs użytkownika. Dla poszczególnych elementów interfejsu użytkownika można zdefiniować unikalne funkcje callback. Funkcja może rozpoznać szereg danych związanych z zdarzeniem, np. czy dotyczy ono przyciśnięcia lewego czy prawego przycisku myszki. Dodatkowo znane są współrzędne x-y pozycji położenia kursora myszki podczas przyciśnięcia jej przycisków. Rozpoznawane są też zdarzenia związane z przesuwaniem lub zmianą rozmiarów paneli interfejsu podczas działania programu. Wejścia z klawiatury są zintegrowane z wejściami z myszki. W przypadku użycia klawiatury istnieje możliwość rozpoznania rodzaju użytego klawisza.

Sposób rozpoznania i reakcji funkcji callback na różnorodne zdarzenia jest bardzo prosty. Prototyp każdej funkcji callback jest ustalany w środowisku LW. Programista określa unikalną nazwę funkcji, ale lista i rodzaj jej argumentów jest stały. Wszystkie dane zdarzenia wygenerowanego przez interfejs użytkownika są przenoszone do indywidualnej funkcji callback przez listę jej argumentów. Przykładowo funkcja callback AcquireData, związana z przyciskiem polecającym wykonanie akwizycji, będzie miała następujący prototyp:

AcquireData (int panel, int control, int event, void * callbackData, int eventData1, int eventData2)

gdzie:

- panel - Panel macierzysty będący źródłem zdarzenia (GUI może mieć kilka paneli macierzystych).
- control – Obiekt sterujący będący źródłem zdarzenia (ta sama funkcja może być przypisana do kilku obiektów sterujących).
- event - Typ wygenerowanego zdarzenia, np. LEFT_CLICK, RIGHT_CLICK lub KEYPRESS.
- callbackData - Dane dodatkowe definiowane przez programistę.
- eventData - Dane charakteryzujące zdarzenie; zależne od typu zdarzenia. Np. dla zdarzenia LEFT_CLICK eventData1 i eventData2 prezentują odpowiednio współrzędne x-y kursora w pikselach. Dla zdarzenia KEYPRESS argumenty zawierają kod ASCII użytego klawisza.

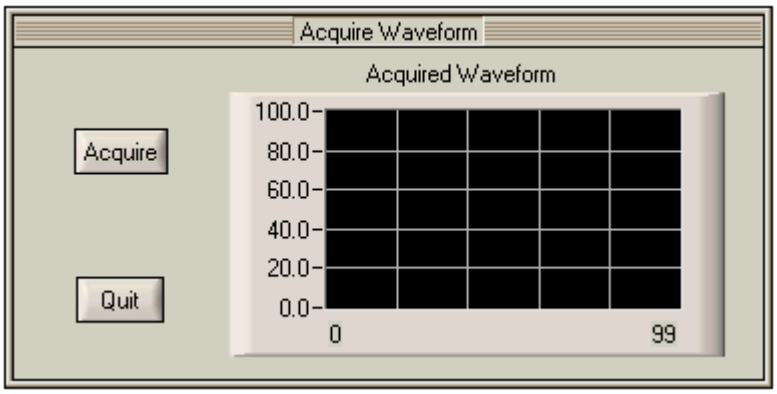
Tabela 1: Typy zdarzeń.

Typ zdarzenia	EvenData1	EvenData2
EVENT_NONE		
EVENT_COMMIT		
EVENT_VAL_CHANGED		
EVENT_LEFT_CLICK	Pozycja Y kursora	Pozycja X kursora
EVENT_LEFT_DOUBLE_CLICK	Pozycja Y kursora	Pozycja X kursora
EVENT_RIGHT_CLICK	Pozycja Y kursora	Pozycja X kursora
EVENT_RIGHT_DOUBLE_CLICK	Pozycja Y kursora	Pozycja X kursora
EVENT_KEYPRESS	Kod użytego klawisza	Wskaźnik do kodu użytego klawisza
EVENT_GOT_FOCUS	Dotychczas aktywny panel sterujący	
EVENT_LOST_FOCUS	Nowy aktywny panel sterujący	
EVENT_CLOSE		
EVENT_PANEL_MOVE		
EVENT_PANEL_SIZE		
EVENT_IDLE		
EVENT_TIMER_TICK	Wskaźnik do aktualnego czasu w sekundach	Wskaźnik do czasu upływającego od ostatniego zdarzenia TIMER_TICK

Kiedykolwiek wystąpi zdarzenie (tab.1) interfejsu użytkownika, wszystkie informacje dotyczące tego zdarzenia są przeniesione do funkcji callback przez jej argumenty. Programista decyduje czy i jak te informacje zostaną wykorzystane. W większości przypadków funkcje callback uwzględniają tylko zdarzenia EVENT_COMMIT, które odpowiada kliknięciu lewym klawiszem myszki na wybranym panelu sterującym lub użyciu klawisza <Return>, gdy panel jest wybrany. W takiej sytuacji tylko to zdarzenie jest ważne dla funkcji callback; pozostałe są ignorowane. LW dostarcza wielu typów zdarzeń i można rozbudować określoną funkcję callback tak, aby działała w sposób uzależniony od rodzaju zdarzenia.

Ogólna idea budowy funkcji callback jest bardzo prosta. Jej ciało jest zwykle obszerną instrukcją wyboru języka C (wyrażenie switch-case lub if-else), która na podstawie rodzaju zdarzenia decyduje co powinno być zrobione. Warunki wyboru określają obsługiwane zdarzenia, czyli ważne zdarzenia dla danej funkcji callback.

Prosty interfejs GUI z rys.1-3 ma dwa obiekty sterujące: przycisk Acquire i Quit oraz jeden obiekt prezentacyjny Acquired Waveform. Z przyciskiem Acquire jest związana funkcja callback AcquireData a z przyciskiem Quit funkcja Shutdown. Zdarzenie COMMIT przycisku Acquire realizuje akwizycję sygnału z urządzenia akwizycji (np. karta AC) oraz prezentuje je na wykresie. To samo zdarzenie przycisku Quit powoduje zakończenie działania programu.



Rys.1-3. Przykład prostego interfejsu aplikacji.

Przedstawiony niżej pseudo-kod ilustruje implementację tej aplikacji:

```
main()
{
    handle = LoadPanel( 0, "demo.uir", PANEL);
    DisplayPanel( handle );
    RunUserInterface();
}

// Funkcje callback
int AcquireData ( int panel, int control, int event, void * callbackData, int eventData1, int eventData2)
{
        if ( event == EVENT_COMMIT)  {
            // Akwizycja danych z urządzenia
            .....
            .....
            PlotY( handle, PANEL_GRAPH, datapoints, ..... );
        }
        return 0;
}

int Shutdown ( int panel, int control, int event, void * callbackData, int eventData1, int eventData2)
{
        if ( event == EVENT_COMMIT)
        QuitUserInterface( 0 );
        return 0;
}
```

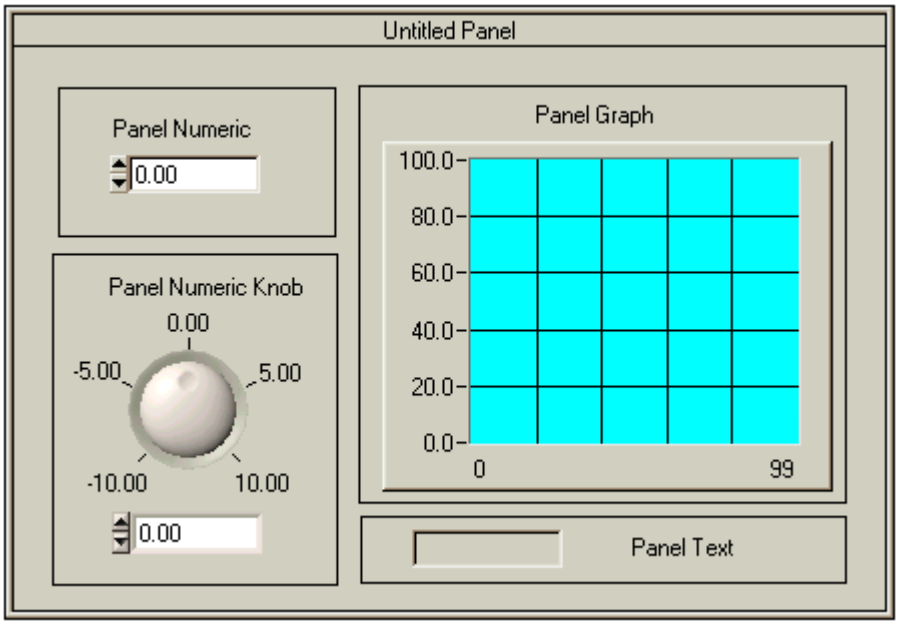
W przypadku rozbudowanego interfejsu użytkownika można w prosty sposób dodać wyświetlanie informacji wyjaśniających przeznaczenie poszczególnych elementów GUI. Ogólnie przyjętą metodą jest kliknięcie prawym klawiszem myszki na danym elemencie interfejsu. W tej sytuacji funkcje callback muszą rozróżnić zdarzenie LEFT_CLICK oraz RIGHT_CLICK i dla przytoczonego przykładu mogą przyjąć postać:

```
int AcquireData ( int panel, int control, int event, void * callbackData, int eventData1, int
eventData2)
{
        switch (event)
        {
         case EVENT_LEFT_CLICK :
            // Akwizycja danych z urządzenia
            .....
            PlotY( handle, PANEL_GRAPH, datapoints, ..... );
            break;
        case EVENT_RIGHT_CLICK :
            // Wyświetlenie informacji wyjaśniającej przeznaczenie przycisku Acquire
            .....
            break;
        }
        return 0;
}

int Shutdown ( int panel, int control, int event, void * callbackData, int eventData1, int eventData2)
{
        switch (event)
        {
         case EVENT_LEFT_CLICK :
        QuitUserInterface( 0 );
            break;
        case EVENT_RIGHT_CLICK :
            // Wyświetlenie informacji wyjaśniającej przeznaczenie przycisku Quit
            .....
            break;
        }
        return 0;
}
```

Obiekty sterujące i prezentacyjne.

Jak wynika z dotychczasowego opisu pewne obiekty interfejsu użytkownika mają przypisaną im funkcję callback a inne nie. Różnica wynika przede wszystkim z przeznaczenia obiektu, a więc czy jest on używany jako sterujący czy prezentacyjny. Inaczej mówiąc, czy jest wejściem czy wyjściem danych dla aplikacji. Wyświetlacz tekstowy, graficzny lub LED prezentują pewne stany systemu, pokazują stany pewnych zmiennych i wyprowadzają informacje z aplikacji, np. LED może służyć do wskazania zajętości systemu akwizycją danych. Ogólnie nie oczekuje się, że użytkownik zechce wprowadzić informacje poprzez takie elementy. Co za tym idzie, obiekty używane do wyprowadzania informacji nie potrzebują własnej funkcji callback, ponieważ nie oczekuje się generacji przez nie zdarzeń użytkownika. Przeciwna sytuacja występuje w odniesieniu do obiektów sterujących takich jak przełączniki, pokrętła nastawcze, przyciski rozkazowe itp. Do przetworzenia zdarzeń związanych z ich użyciem potrzebne są odpowiednie funkcje callback przypisane do nich. Bez takich funkcji użytkownik nie ma możliwości oddziaływania na aplikację, ponieważ wszystkie istotne rzeczy są realizowane w tych funkcjach.



Rys.1-4. Przykładowe obiekty interfejsu; po lewej sterujące, po prawej prezentacyjne.

W ogólnym opisie obiektów interfejsu użytkownika nie określa się wyraźnie ich przydziału do jednej z wspomnianych kategorii. Powodem tego jest fakt, że większość obiektów nie ma wrodzonych własności nadających im charakter sterujący lub prezentacyjny. Wiele z nich można używać na oba sposoby, np. obiekt przełącznika może pracować jako element prezentacyjny (jego ustawienie wskazuje pewien stan systemu). Wszystko zależy od przyjętej koncepcji rozwiązania interfejsu użytkownika, konfiguracji i wyposażenia obiektów w funkcje callback. Dla użytkownika końcowego aplikacji jest jednak istotne, aby obiekty funkcjonowały zgodnie z ogólnie przyjętymi oczekiwaniami (zgodnymi z intuicją).

Kreując obiekt, projektant musi nadać mu charakter sterujący lub prezentacyjny. Realizuje to za pomocą okna edycji własności obiektu (rys.4-4) ustawiając odpowiednio pole *Control Mode*. Pole to pozwala ustawić jeden z czterech dostępnych trybów pracy obiektu:

- *Normal* – Po zmianie stanu obiekt generuje zdarzenie VAL_CHANGED.
- *Hot* – Po zmianie stanu obiekt generuje zdarzenie COMMIT.
- *Validate* – Po zmianie stanu obiekt sprawdza czy wartość obiektu mieści się w ustalonym dla niego zakresie (przy aktywnej opcji kontroli zakresu) i generuje zdarzenie COMMIT.
- *Indicator* – Użytkownik nie może zmienić stanu obiektu i nie generuje on ani zdarzenia COMMIT ani VAL_CHANGED.

Pierwsze trzy tryby pracy nadają obiektowi charakter sterujący. Domyślnie jest ustawiony tryb *hot*. Charakter prezentacyjny obiektu zapewnia tryb *indicator*.

Podstawowe elementy pakietu LabWindows/CVI.

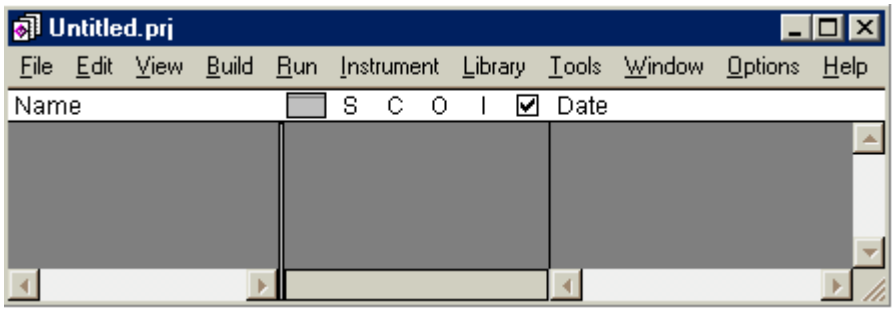
Pakiet LW składa się z trzech składników - okna projektu, edytora UIR oraz kompilatora C:

- **Okno projektu** – pozwala utworzyć listę plików źródłowych, bibliotek oraz drajwerów składających się na projektowaną aplikację.
- **Edytor UIR** – służy do edycji interfejsu użytkownika aplikacji. Tworzy plik *uir*, który jest plikiem opisującym zasoby interfejsu. Edytor dysponuje biblioteką obiektów, z których buduje się interfejs graficzny.
- **Kompilator C** – służy do prezentacji, modyfikacji, edycji, kompilacji i wykonywania kodu kreowanej aplikacji.

4. Projektowanie aplikacji w LabWindows; przykład 1.

Przykład wprowadzający jest bardzo prosty i zawiera wyłącznie panel macierzysty z jednym obiektem w postaci przycisku rozkazowego, który służy do zakończenia działania aplikacji. Choć jest bardzo prostym przykładem, pokazuje wszystkie niezbędne czynności potrzebne do wykreowania aplikacji.

Okno projektu. Po uruchomieniu LW, pokazuje się okno projektu wyświetlające listę plików ostatnio realizowanego projektu. W celu utworzenia nowego projektu należy wybrać z menu okna projektu pozycję *File/New/Project*. W procesie kreowania nowego projektu należy potwierdzić wyładowanie aktualnego projektu oraz ustalić opcje nowego projektu. W wyniku uzyskuje się puste okno projektu zatytułowane uogólnioną nazwą *untitled.prj* (rys.4-1).

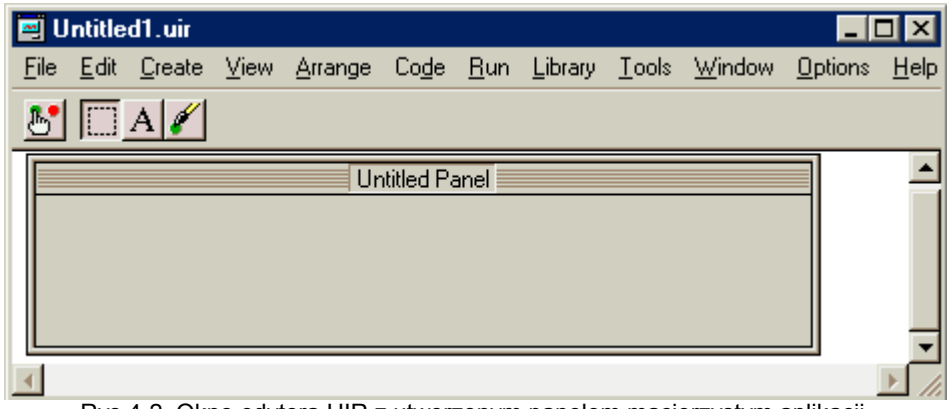


Rys.4-1. Okno projektu.

Sposób tworzenia aplikacji. Najlepszym sposobem tworzenia aplikacji w LW jest utworzenie w pierwszej kolejności jej interfejsu użytkownika. Na jego podstawie zostanie wytworzony szkielet kodu programu w C, wymagający jeszcze pewnych szczegółowych uzupełnień. Od autora aplikacji wymaga się zatem ustalenia w pierwszej kolejności wszystkich wejść i wyjść aplikacji, postaci elementów sterujących i prezentacyjnych oraz decyzji, które z nich zostaną wyposażone w funkcje callback, jakie będą zadania tych funkcji oraz jakie zdarzenia będą obsługiwały. W przypadku rozpatrywanego projektu ustalenia są natychmiastowe: jeden przycisk rozkazowy o nazwie *Quit* wyposażony w funkcję *callback* o nazwie *Bye* obsługującą zdarzenie COMMIT i realizującą zakończenie działania programu.

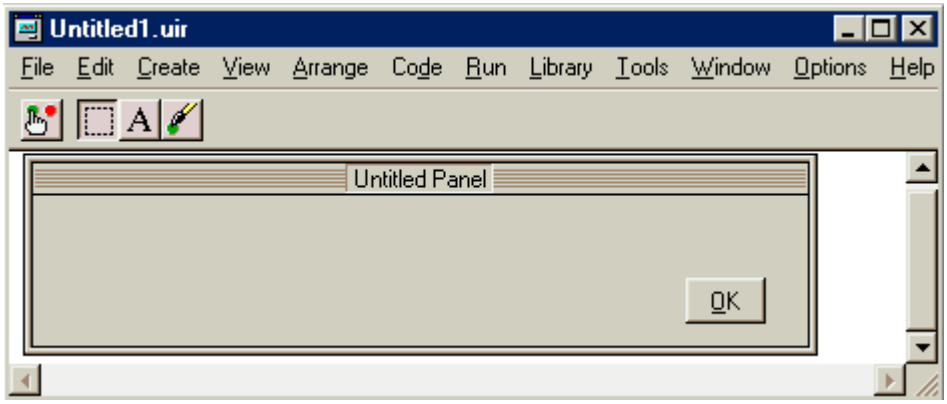
Każda aplikacja kreowana w LW wymaga elementu wymuszającego jej zakończenie. Jeśli pominie się element kończący działanie programu, przerwanie działania aplikacji jest niemożliwe, pozostaje tylko wyłączenie komputera. Dlatego dodanie elementu kończącego do interfejsu użytkownika jest konieczne. Do okna tworzonej aplikacji są automatycznie dodawane trzy windows'owe elementy sterujące: minimalizacja i maksymalizacja okna oraz zamknięcie aplikacji. Niestety standardowy element zamykania aplikacji działa dopiero po wprowadzeniu obiektu kończącego, zdefiniowaniu związanej z nim funkcji kończącej i odpowiednim skonfigurowaniu własności panelu macierzystego (pole Close Control okna edycji własności panelu).

Edytor UIR. Interfejs użytkownika tworzy się przy użyciu *edytora UIR*. Jego uruchomienie realizuje się po wybraniu pozycji *File/New/User Interface* z menu okna projektu. Otwarte okno *edytora UIR* domyślnie tworzy panel macierzysty, stanowiący okno kreowanej aplikacji (rys.4-2). Jeśli opcja domyślnego tworzenia jest wyłączona można wykreować panel macierzysty przy użyciu pozycji *Create/Panel* menu edytora.



Rys.4-2. Okno edytora UIR z utworzonym panelem macierzystym aplikacji.

Dodanie elementu sterującego w postaci przycisku rozkazowego do panelu macierzystego wymaga wybrania pozycji Create/Command Button z menu edytora UIR (rys.4-3).

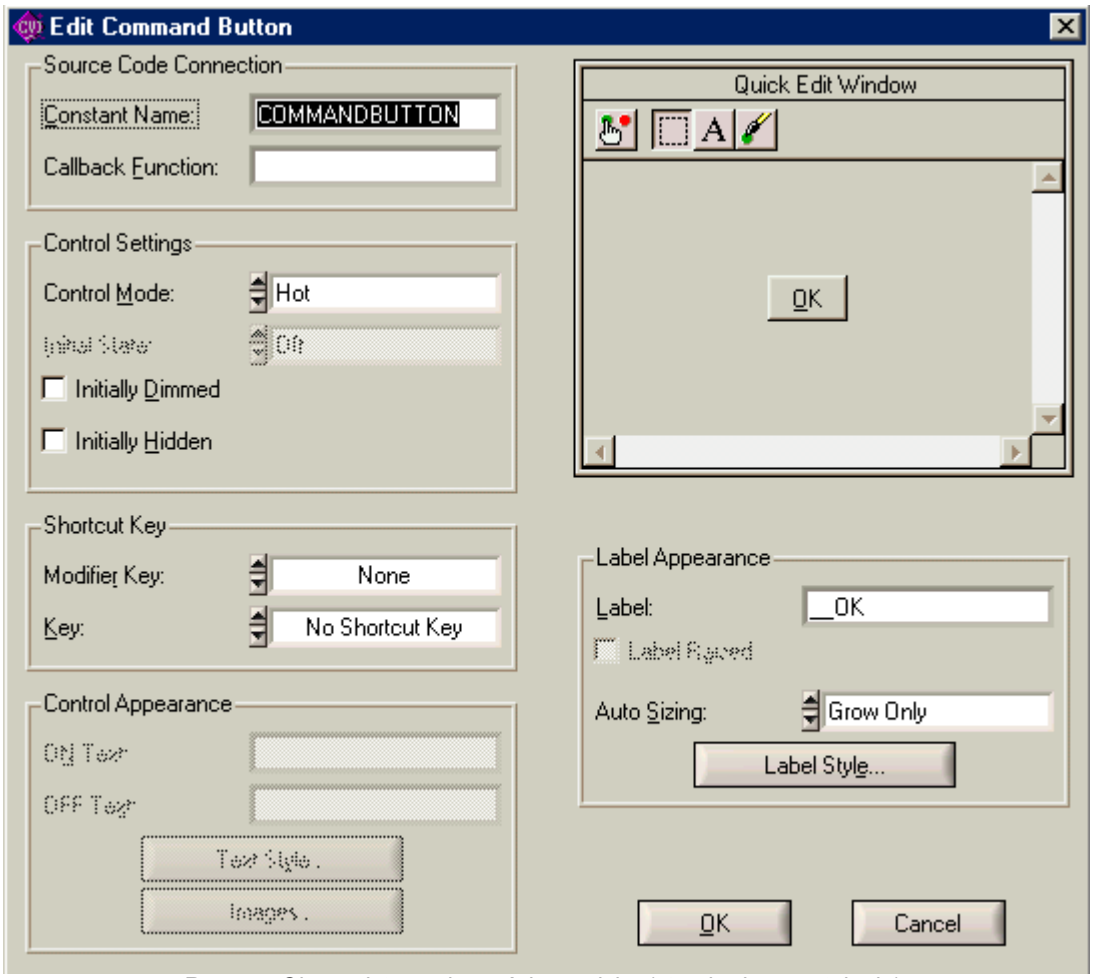


Rys.4-3. Panel macierzysty aplikacji z dodanym przyciskiem rozkazowym.

Każdy element interfejsu użytkownika ma określone własności (nazwę, etykietę, kolor itp.). Własności te można konfigurować stosownie do potrzeb przy użyciu edytora własności panelu, który uruchamia się podwójnie przyciskając lewy klawisz myszki ustawionej na danym elemencie interfejsu. Dla wprowadzonego przycisku rozkazowego określone zostaną trzy własności: nazwa panelu, nazwa funkcji callback stowarzyszonej z przyciskiem oraz etykieta przycisku (rys.4-4).

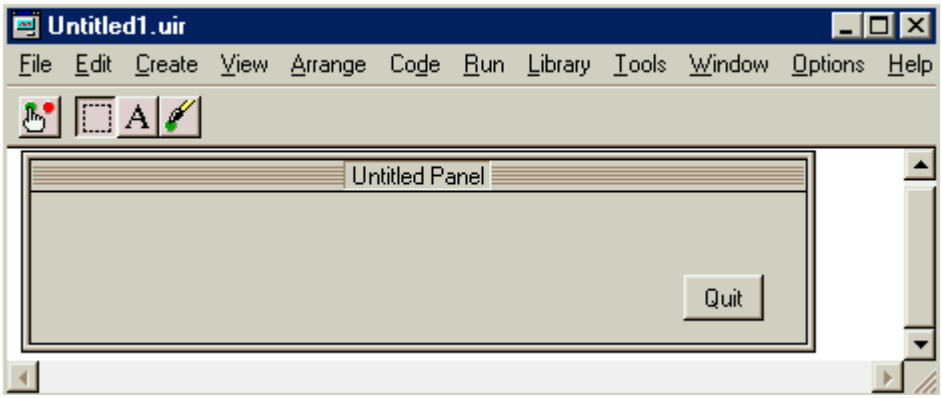
Jak już zaznaczono wcześniej każdy obiekt jest charakteryzowany stałą o unikalnej nazwie (Constant name). Podczas wstawiania obiektu do budowanego interfejsu LW kreuje nazwy domyślne, w tym przypadku jest to COMMANDBUTTON. Jeśli interfejs ma mieć więcej elementów tego typu, następne otrzymają nazwy COMMANDBUTTON1, COMMANDBUTTON1 itd. Projektant może zmodyfikować te nazwy. Modyfikacja musi zachować unikalność nazw natomiast jej podstawowym celem jest jednoznaczne wskazanie przez nazwę przeznaczenia elementu sterującego (czytelność kodu programu). W przykładzie przyciskowi została przypisana nazwa QUITBUTTON (wpis w polu Constant name).

Przycisk ma kończyć działanie aplikacji, zatem podobnie jak każdy element sterujący interfejsu wymaga funkcji callback. W polu Callback Funktion trzeba wpisać unikalną nazwę funkcji obsługującej zdarzenia generowane przez przycisk. W przykładzie przyjęto nazwę Bye dla funkcji callback przycisku (wpis w polu Callback Funktion).



Rys.4-4. Okno edytora własności przycisku (standardowe ustalenia).

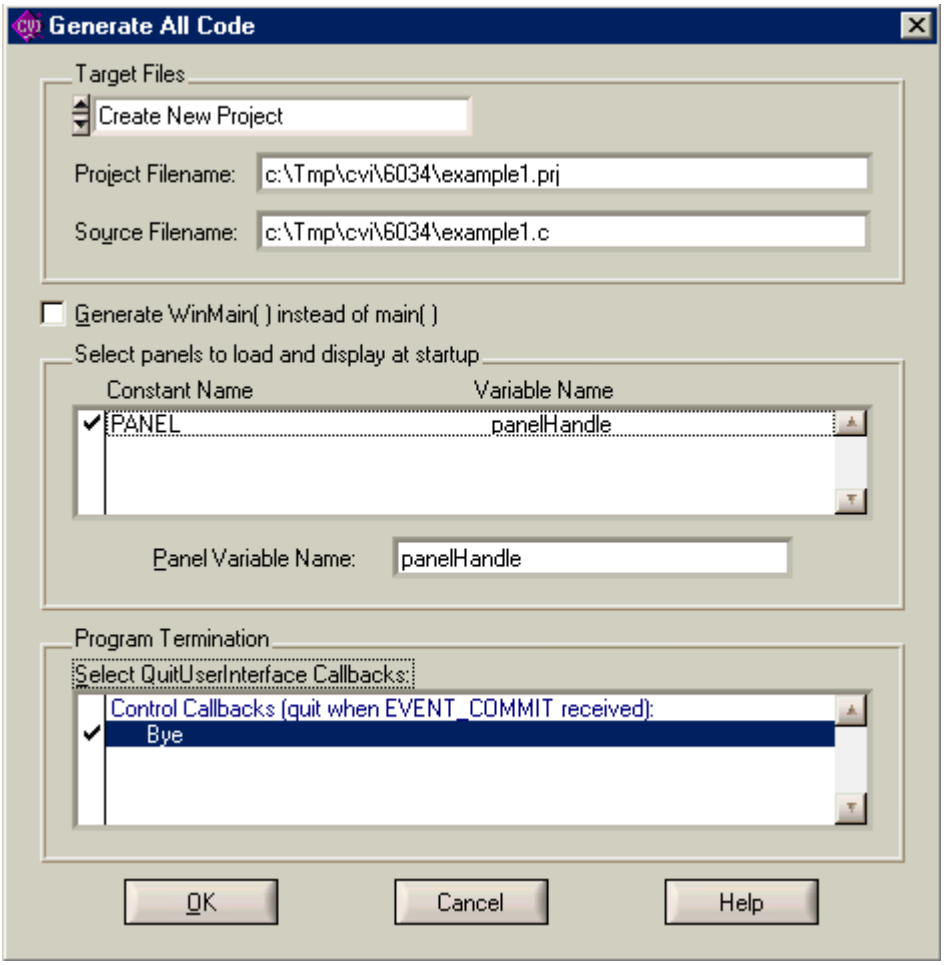
Ostatnią modyfikowaną własnością jest etykieta przycisku, czyli napis występujący na nim. Zmieniono domyślną nazwę OK na Quit, która lepiej oddaje przeznaczenie przycisku (wpis w polu Label). Z przytoczonych trzech modyfikacji tylko ustalenie nazwy funkcji callback jest bezwzględnie potrzebne, pozostałe zmiany mają charakter kosmetyczny.



Rys.4-5. Panel interfejsu użytkownika po modyfikacji własności przycisku.

Generacja szkieletu kodu C programu. W podany wyżej sposób został zaprojektowny interfejs użytkownika. Nie ma jednak jeszcze wygenerowanego żadnego kodu, który mógłby wykonać ten interfejs jak również zaimplementować jego cechy funkcjonalne. Do wygenerowania kodu używa się również edytora UIR. Proces generacji kodu dostarcza wyłącznie szkielet kodu. W pewnym sensie jest on kompletny, ponieważ zapewnia on wykonanie programowe interfejsu użytkownika oraz zapewnia strukturalizację zdarzeń dla funkcji callback. Dla każdej zadeklarowanej w własnościach obiektów

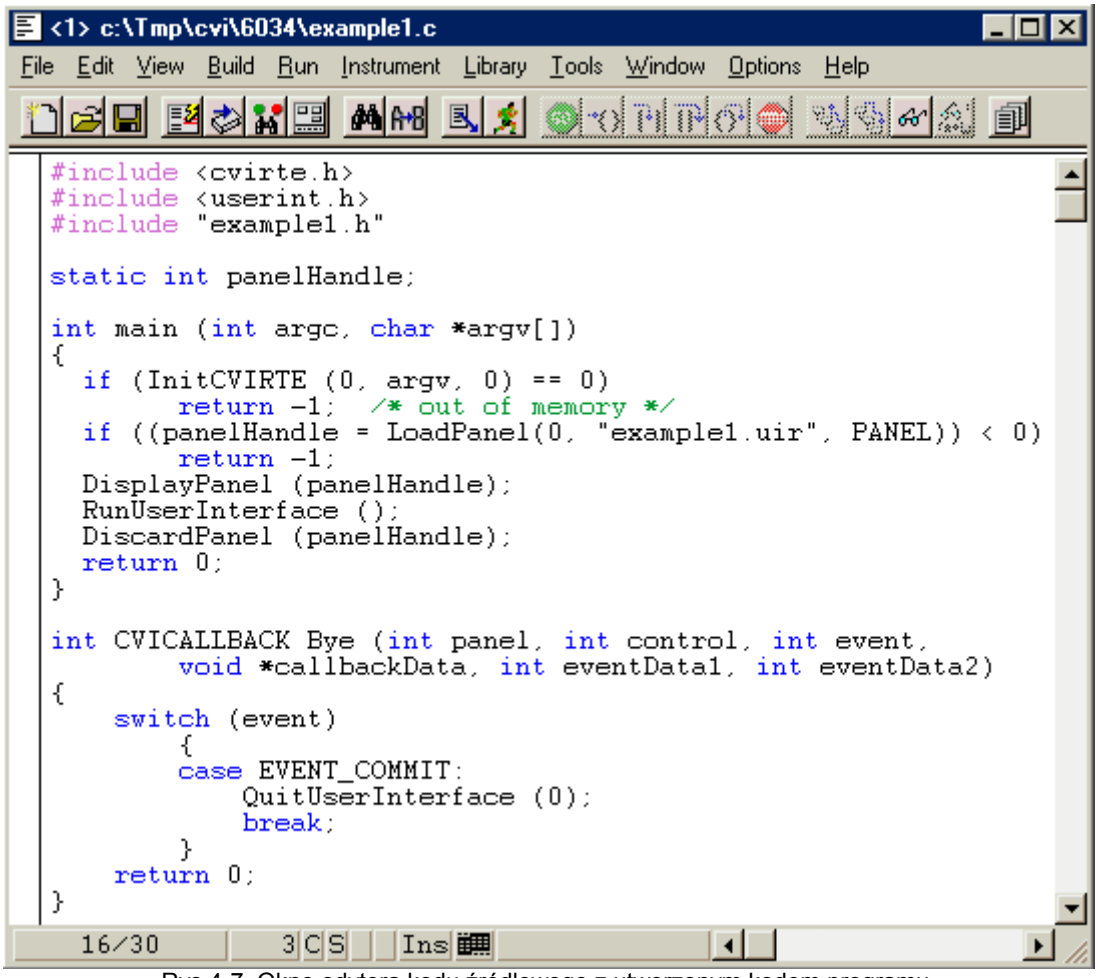
funkcji callback jest utworzony tylko szkielet jej kodu. Zatem aplikacja ma już postać zdolną do działania, ale tylko w zakresie wytworzenia interfejsu. Brak jest jeszcze określenia jej funkcjonalności, co uzyska się dopiero po szczegółowym zdefiniowaniu ciał funkcji callback, czyli zapisie kodu operacji realizowanych w odpowiedzi na konkretne zdarzenia.



Rys.4-6. Panel ustalenia opcji generacji kodu.

Przed przystąpieniem do wygenerowania kodu aplikacji trzeba zapisać projekt pod wybraną nazwą (pozycja *File/Save As* menu edytora UIR). Dopiero teraz można wybrać pozyję *Code/Generate All Code* menu edytora UIR w celu wygenerowania szkieletu kodu aplikacji. Środowisko LW wyświetla planszę pozwalającą ustalić pewne warunki generacji kodu (rys.4-6), między innymi można określić funkcję callback, która realizuje zakończenie działania aplikacji. Tutaj należy zaznaczyć funkcję Bye w sekcji *Program Termination* (jest to jedyna funkcja callback projektowanej aplikacji). Dzięki temu kod funkcji kończącej będzie w zasadzie kompletny, ponieważ generator kodu wpisze do ciała tej funkcji wywołanie funkcji *QuitUserInterface*.

Po przyciśnięciu przycisku OK planszy pokazanej na rys.4-6, generator kodu wyprodukuje szkielet kodu C aplikacji zapisany w pliku z rozszerzeniem .c (tutaj example1.c). Jednocześnie uruchomiony zostanie kompilator C środowiska LW (rys.4-7), który w oknie edytora tekstowego przezentuje kod aplikacji i pozwala uzupełnić funkcje callback w unikalne fragmenty kodu realizujące zasadnicze funkcje projektowanej aplikacji.

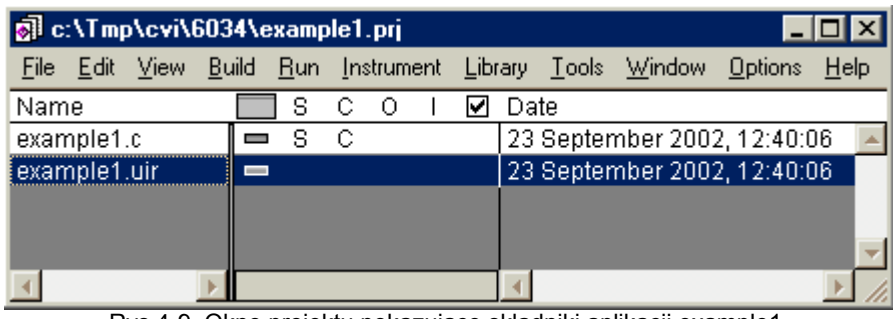


Rys.4-7. Okno edytora kodu źródłowego z utworzonym kodem programu.

W przypadku tego przykładu kod nie wymaga już żadnych uzupełnień. Pozostaje tylko sprawdzenie działania kodu. Kompilację programu uruchamia pozycja *Build/Compile File* menu kompilatora a wykonanie pozycja *Run/Debug example1_dbg.exe* (wersja z możliwością debugowania, można ją wyłączyć).

Rys.4-8. Okno działającej aplikacji.

Po utworzeniu kodu programu okno projektu pokazuje pliki składowe zbudowanej aplikacji (rys.4-9). Poprzez dwukrotne kliknięcie na wybranej pozycji listy składników można automatycznie uaktywnić jej edycję, tutaj wejść do edytora UIR lub edycji kodu źródłowego.



Rys.4-9. Okno projektu pokazujące składniki aplikacji example1.

Kod programu. W celu łatwiejszej analizy kodu warto jest podzielić go na trzy segmenty. Pierwszy segment pokazany na rys.4-11 zawiera dołączane pliki oraz deklaracje zmiennych globalnych. Z trzech dołączanych plików ostatni jest szczególnie istotny, ponieważ zawiera on informacje dotyczące interfejsu użytkownika takie jak definicje stałych i prototypy funkcji callback. Plik nagłówkowy aplikacji (tutaj example1.h) jest tworzony automatycznie podczas kreowania kodu aplikacji (rys.4-10). Projektant nie powinien samodzielnie modyfikować treści tego pliku.

```
/*        ******************   example1.h   ********************************      */
/* LabWindows/CVI User Interface Resource (UIR) Include File                      */
/* Copyright (c) National Instruments 2002. All Rights Reserved.                  */
/* WARNING: Do not add to, delete from, or otherwise modify the contents of       */
/*               this include file.                                               */
/*       ************************************************************************ */
#include <userint.h>
#ifdef __cplusplus
    extern "C" {
#endif

     /* Panels and Controls: */
#define  PANEL                           1
#define  PANEL_QUITBUTTON                2       /* callback function: Bye */

     /* Menu Bars, Menus, and Menu Items: */

       /* (no menu bars in the resource file) */

     /* Callback Prototypes: */
int  CVICALLBACK Bye(int panel, int control, int event, void *callbackData, int eventData1, int eventData2);

#ifdef __cplusplus
    }
#endif
```
Rys.4-10. Treść pliku nagłówkowego aplikacji example1.

Ostatnie wyrażenie pierwszego segmentu deklaruje zmienną globalną panelHandle. Zmienna pamięta wartość numeryczną przypisaną przez system Windows panelowi macierzystemu aplikacji.

```
#include <cvirte.h>        /* Potrzebny, jeśli używa się zewnętrznego kompilatora; może być w innej sytuacji */
#include <userint.h>
#include "kas2.h"

static int panelHandle;
```
Rys.4-11. Pierwszy segment kodu aplikacji example1.

Drugi segment programu jest funkcją główną (rys.4-12). Argumenty funkcji main są nieistotne, można je zlikwidować. Podobnie mało istotne jest pierwsze wyrażenie warunkowe if, potrzebne w przypadku wykorzystywania zewnętrznego kompilatora. Uwagi wymagają natomiast kolejne linie kodu:

- Funkcja LoadPanel ładuje plik example1.uir, opisujący zasoby interfejsu aplikacji.
- Funkcja DisplayPanel wyświetla na pulpicie graficzną postać interfejsu użytkownika (okno aplikacji).

- Funkcja RunUserInterface uruchamia procedurę obsługi zdarzeń interfejsu użytkownika, która kieruje wszystkie zdarzenia użytkownika do właściwej funkcji callback. Proszę zauważyć, że programista nie realizuje ‘ręcznie’ tej procedury, jest ona tworzona automatycznie a jej kod nie jest ujawniony. Program znajduje się w funkcji RunUserInterface aż do momentu wywołania w jednej z funkcji callback funkcji kończącej QuitUserInterface.
- Funkcja DiscardPanel usuwa panel (okno aplikacji) z pamięci i z pulpitu systemu Windows.
- Wyrażenie return kończy program.

```
int main (int argc, char *argv[])
{
        if (InitCVIRTE (0, argv, 0) == 0)
                return -1;          /* out of memory */

        if ((panelHandle = LoadPanel (0, "example1.uir", PANEL)) < 0)
                return -1;
        DisplayPanel (panelHandle);
        RunUserInterface ();
        DiscardPanel (panelHandle);
        return 0;
}
```

Rys.4-12. Drugi segment kodu aplikacji example1.

Trzeci segment kodu jest funkcją callback o nazwie Bye. Funkcja jest wywołana przez funkcję RunUserInterface, gdy wystąpi zdarzenie związane z przyciskiem Quit. Jej argumenty dostarczają informacje o zaistniałym typie zdarzenia. Ciało funkcji zawiera wyrażenie switch-case, które filtruje zdarzenia. Filtracja zapewnia, że tylko zdarzenie COMMIT spowoduje wywołanie funkcji QuitUserInterface, która przerwie działanie funkcji RunUserInterface i w efekcie skończy się działanie aplikacji.

```
int CVICALLBACK Bye (int panel, int control, int event, void *callbackData, int eventData1, int eventData2)
{
        switch (event)
                {
                case EVENT_COMMIT:
                        QuitUserInterface (0);
                        break;
                }
        return 0;
}
```

Rys.4-13. Trzeci segment kodu aplikacji example1.

Podczas generacji kodu funkcji callback, pomiędzy etykietę EVENT_COMMIT a wyrażenie break wprowadzane są puste linie wskazujące na potrzebę wpisania przez projektanta kodu realizującego żądane operacje. Tutaj umiejscowione są kody realizujące zasadnicze zadania aplikacji i tu, pomijając utworzenie interfejsu użytkownika, mieści się zasadnicza praca projektanta aplikacji. W przypadku funkcji Bye wytworzony kod był kompletny, ponieważ w procesie generacji zaznaczono ją jako funkcję kończącą aplikację. W pewnych projektach funkcja kończąca może wymagać szerszych akcji, np. zwolnienia pewnych zasobów i wtedy trzeba jej kod rozszerzyć ‘ręcznie’.

Podsumowanie. Projektowanie aplikacji w LW realizuje się w trzech kolejnych etapach:
- Projekt interfejsu użytkownika.
- Generacja szkieletu kodu aplikacji.
- Dodanie kodów wykonawczych do wszystkich funkcji callback.

5. Projektowanie aplikacji w LabWindows; przykład 2.

Przykład 2 jest rozszerzeniem poprzedniego projektu i ilustruje wprowadzanie danych do aplikacji oraz prezentację danych uzyskanych w wyniku jej działania. Aplikacja dysponuje zmienną, której wartość jest inkrementowana po każdym przełączeniu przełącznika dwupozycyjnego. Aktualną wartość zmiennej prezentuje wyświetlacz numeryczny. Interfejs użytkownika aplikacji musi być

rozbudowany w stosunku do przykładu pierwszego o dwa dodatkowe elementy: element nastawczy w postaci przełącznika binarnego oraz element prezentacyjny w postaci pola numerycznego.

Dostępne elementy nastawcze.

LW dysponuje różnymi typami obiektów, które można zastosować jako elementy nastawcze interfejsu użytkownika. Każdy z typów występuje w różnych wersjach graficznych. Bardzo często stosowanymi elementami sterującymi są:

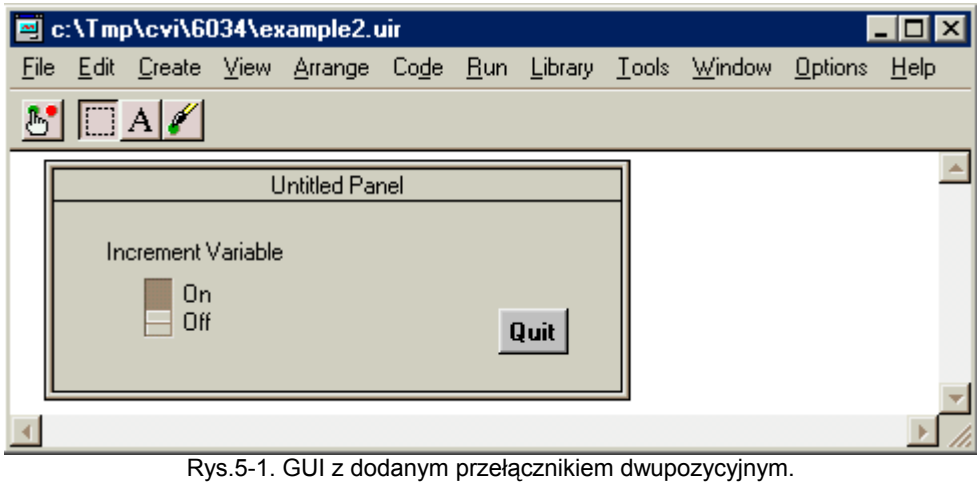
- Command Button – przycisk do inicjowania określonych działań.
- Toggle Button – klucz dwustanowy; wciśnięty dostarcza wartośc 1, wyciśnięty wartość 0.
- Binary Switch – przełącznik dwustanowy on/off; stanom można przypisać wartości.
- Numeric – Służy do wprowadzania wartości liczbowych.
- Text – Służy do wprowadzania tekstów alfanumerycznych.
- Ring – Wybór jednej z dostępnych pozycji. Każda pozycja jest określona etykietą i wartością; etykieta jest tekstem a wartość liczbą lub stringiem.
- List Box – Wybór pozycji z listy pozycji. Każda pozycja listy jest określona etykietą i wartością; etykieta jest tekstem a wartość liczbą lub stringiem.

Dodanie przełącznika dwustanowego do GUI aplikacji.

Po uruchomieniu LW otwiera się okno projektu. Należy otworzyć poprzedni projekt i przejść do edytora UIR. Po uaktywnieniu panelu macierzystego (kliknięcie w obszarze jego okna), za pomocą pozycji Create/Binary Switch menu edytora kreuje się przełącznik dwupozycyjny (należy wskazać żądaną postać graficzną przełącznika). Teraz należy określić żądane własności elementu sterującego (podwójne kliknięcie na konfigurowanym elemencie). W tym przypadku należy:

- Określić nazwę funkcji callback związanej z przełącznikiem, np. *FunctionIncrement*.
- Etykietę przełącznika zmienić na postać opisującą jego przeznaczenie, np. *Increment Variable*.

Pozostałe własności mogą pozostać bez zmian, w tym przypisane wartości liczbowe do stanów on/off przełącznika (tutaj odpowiednio 1 i 0). Projektant interfejsu ma możliwość wizualnego sprawdzenia wyglądu i działania elementów sterujących interfejsu użytkownika na etapie projektowania. W tym celu wystarczy użyć pozycji *Options/Operate Visible Panels* menu edytora UIR.



Rys.5-1. GUI z dodanym przełącznikiem dwupozycyjnym.

Dodanie szkieletu kodu nowej funkcji callback.

W poprzednim przykładzie generowano szkielet całego kodu aplikacji. Teraz, zakładając rozwinięcie przykładu pierwszego, wystarczy dołączyć tylko kod dla nowej funkcji callback o nazwie *Function Increment*. W tym celu, po zapisaniu pliku interfejsu użytkownika, najpierw trzeba wskazać plik źródłowy do, którego kod nowej funkcji ma być dodany. Realizuje się to za pomocą pozycji Code/Set Target File menu edytora UIR. Teraz można kliknąć prawym klawiszem myszki na przełączniku i z rozwiniętego menu wybrać pozycję *Generate Control Callbacks*. Dotychczasowy kod aplikacji zostanie w sekcji trzeciej uzupełniony dodatkową funkcją callback:

```
int CVICALLBACK FunctionIncrement (int panel, int control, int event,
                                   void *callbackData, int eventData1, int eventData2)
{
        switch (event)
                {
                case EVENT_COMMIT:    // Tutaj wpisać kod operacji realizowanych po zdarzeniu !

                        break;
                }
        return 0;
}
```

Rys.5-2. Szkielet funkcji obsługującej zdarzenia przełącznika dwupozycyjnego.

Dodanej funkcji brakuje żądanej funkcjonalności. Brak jej specyficznego dla danej aplikacji kodu po etykiecie *case EVENT_COMMIT*, który projektant aplikacji musi utworzyć samodzielnie przechodząc do edytora kompilatora środowiska LW. W tym przykładzie należy dodać:

- W sekcji pierwszej deklarację zmiennej rejestrującej każde użycie przełącznika, np. **static int x;**
- W funkcji FunctionIncrement operację inkrementacji tej zmiennej, np. **x++;**

Na tym etapie aplikacji brakuje tylko prezentacji aktualnej zawartości zmiennej rejestrującej x. Do tego celu trzeba ją wyposażyć w wyświetlacz numeryczny.

Dostępne elementy prezentacyjne.

Panele prezentacyjne nie generują zdarzeń i wobec tego nie potrzebują własnych funkcji callback. Ich uaktualnianie jest realizowane przez funkcje callback elementów sterujących interfejsu użytkownika. LW dysponuje następującymi typami paneli prezentacyjnych:

- LED – Dioda świecąca wskazująca stany on/off.
- Numeric – Wyświetlacz danych numerycznych.
- Text – Wyświetlacz danych alfanumerycznych.
- Graph – Wykresy.

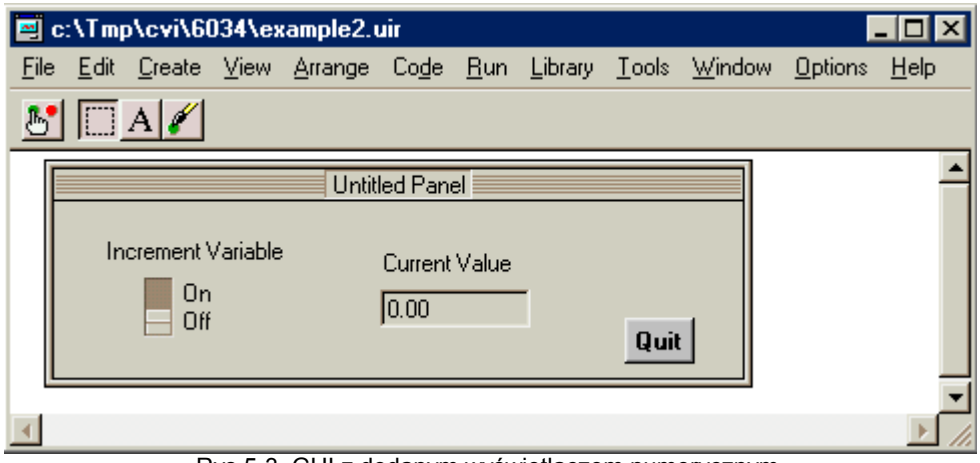
Na liście własności każdego panelu istnieje pozycja określająca tryb jego pracy, która decyduje o tym czy pracuje on jako element nastawczy (*Hot*) czy prezentacyjny (*Indicator*). Elementy typu *Numeric* oraz *Text* domyślnie są wykorzystywane jako elementy wprowadzania danych i po utworzeniu takiego obiektu jest on domyślnie ustawiony w trybie *Hot*. Jeżeli ma on służyć jako wyświetlacz należy po jego wykreowaniu ustawić tryb *Indicator*. Panele *LED* i *Graph* domyślnie są ustawiane w trybie prezentacji danych.

Dodanie wyświetlacza numerycznego.

Dodanie do interfejsu użytkownika aplikacji wyświetlacza numerycznego odbywa się tak samo jak przełącznika dwupozycyjnego. Po jego wykreowaniu trzeba jeszcze odpowiednio go skonfigurować:

- Ustawić tryb prezentacyjny; pole Control Mode ustawić w pozycji Indicator.
- Ustawić typ danych zgodny z typem wartości dostarczanych do wyświetlacza; tutaj całkowity int (zmienna x jest typu int).
- Etykietę wyświetlacza zmodyfikować do postaci sugerującej rodzaj prezentowanych danych.

Z wymienionych modyfikacji tylko dwie pierwsze są niezbędne, trzecia ma charakter kosmetyczny. Interfejs aplikacji jest już gotowy. Potrzebna jest jeszcze drobna modyfikacja kodu aplikacji, polegająca na przekazaniu do wyświetlacza aktualnej wartości zmiennej x po każdym użyciu przełącznika.



Rys.5-3. GUI z dodanym wyświetlaczem numerycznym.

Uzupełnienie kodu aplikacji.

Każdy z elementów sterujących czy prezentacyjnych charakteryzuje się swoją wartością. Element sterujący ma wartość wprowadzoną ostatnio przez użytkownika a element prezentacyjny wartość wprowadzoną przez aplikację. Dostęp programowy do wartości obiektów sterujących i prezentacyjnych odbywa się przy użyciu funkcji SetCtrlVal (ustaw) i GetCtrlVal (dostarcz):

int GetCtrlVal (int panelHandle, int controlID, void *value);

int SetCtrlVal (int panelHandle, int controlID, ...);

gdzie:

- PanelHandle – Specificator panelu macierzystego.
- ControlID – Zdefiniowana stała, która jest przypisana do elementu interfejsu użytkownika.
- *value – wskaźnik do zmiennej poprzez, którą zostanie zwrócona aktualna wartość obiektu. Typ musi odpowiadać typowi danych obiektu sterującego/prezentacyjnego.
- ... – lista wartości przekazywanych do obiektu sterującego/prezentacyjnego.
- Wartość zwrócona prezentuje status wykonania funkcji.

Uzupełnienie kodu aplikacji polega na zmodyfikowniu funkcji FunctionIncrement tak, aby po inkrementacji zmiennej x uaktualnić wyświetlacz numeryczny za pomocą funkcji SetCtrlVal. Kod tej funkcji po uzupełnieniu ma postać:

```
int CVICALLBACK FunctionIncrement (int panel, int control, int event,
                                   void *callbackData, int eventData1, int eventData2)
{
        switch (event)
                {
                case EVENT_COMMIT:    // Tutaj wpisać kod operacji realizowanych po zdarzeniu !
                        x++;
                        SetCtrlVal( panelHandle, PANEL_NUMERIC, x );
                        break;
                }
        return 0;
}
```

Rys.5-4. Funkcja obsługująca zdarzenia przełącznika dwupozycyjnego.

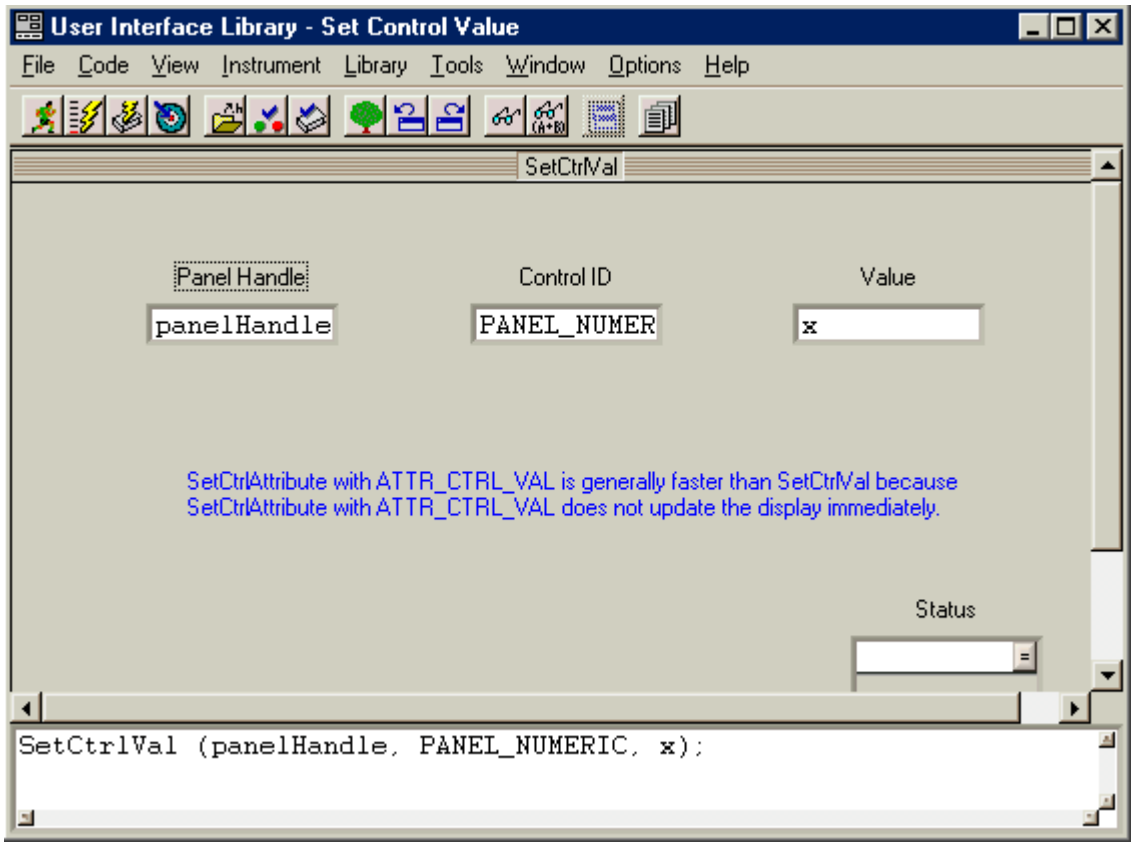
Ułatwienia w edycji wywołań funkcji bibliotecznych.

Funkcje SetCtrVal i GetCtrlVal są funkcjami bibliteki, którą dysponuje środowisko LW. LW posiada setki funkcji zgrupowanych w 16 bibliotekach: User Interface, Advanced Analisis, Data Acquisition, VXI, GPIB, RS232, VISA, TCP itd. Zapamiętanie ich nazw, listy argumentów i wartości zwracanych jest praktycznie niemożliwe. Stąd edycja ich wywołań jest bardzo niewygodna. Środowisko edycyjne kompilatora zostalo wyposażone w mechanizm ułatwiający generację wywołań funkcji bibliotecznych. Proces kreowania wywołania funkcji polega na wyborze żądanej funkcji z listy dostępnych. W wyniku tego wyboru pojawia się plansza edycji wywołania funkcji, która zawiera pola edycyjne dla każdego argumentu funkcji oraz tymczasowe okno z zapisem wywołania funkcji. Zadaniem projektanta jest

określenie treści pól argumentów oraz wartości zwracanej. Pola argumentów mogą zawierać stałe wartości lub zmienne. W przypadku zmiennych istnieje możliwość rozwinięcia listy zmiennych programu i wybrania zmiennej, która ma być użyta jako jeden z argumentów funkcji. Jest to niewątpliwe bardzo przydatne w sytuacji rozbudowanych aplikacji, kiedy zapamiętanie wszystkich zmiennych jest kłopotliwe. W miarę wypełniania pól argumentów w oknie tymczasowego zapisu wywołania funkcji, automatycznie generuje się żądana postać instrukcji programu. Po wykreowaniu wywołania wystarczy wywołać operację przekopiowania instrukcji do kodu źródłowego aplikacji. Instrukcja zostanie wstawiona w miejscu aktualnego położenia kursora edytora tekstu programu.

W odniesieniu do funkcji SetCtrlVal wprowadzanej w funkcji callback FunctionIncrement operacja wytworzenia jej wywołania przedstawia się następująco:

- Kursor edytora tekstu programu należy ustawić w linii po instrukcji x++; wewnątrz funkcji FunktionIncrement.
- Wybrać z pozycji Library menu edytora biblitekę User Interface i odszukać żądaną funkcję. W przypadku funkcji SetCtrlVal trzeba przejść w strukturze drzewiastej biblioteki ścieżkę: *Library/User Interface/Controls, Graphs,.../General Function*. Rozbudowana struktura drzewiasta biblitek jest pewnym mankamentem podczas poszukiwania żądanej funkcji.
- Po wyborze funkcji pojawia się okno edycji wywołania funkcji. W przypadku funkcji SetCtrlVal posiada ono trzy pola argumentów i pole wartości zwracanej. Jeśli nie wykorzystuje się wartości zwracanej pole to można pozostawić puste. Pola argumentów należy wypełnić wartościami stałymi lub nazwami zmiennych.



Rys.5-5. Panel edycji wywołania funkcji.

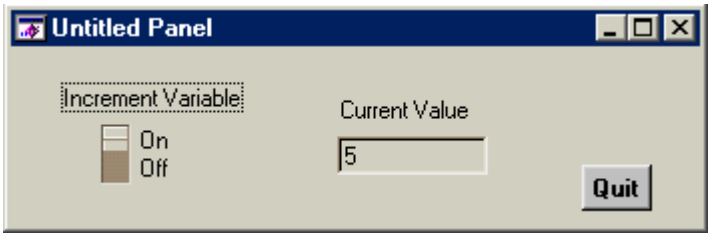
- Do pola PanelHandle należy wprowadzić nazwę zmiennej zawierającej specyfikator panelu macierzystego elementu interfejsu, którego funkcja dotyczy. Można wprost wpisać nazwę tej zmiennej lub jeśli projektant jej nie pamięta, może posłużyć się metodą alternatywną:
  - Uaktywnić pole edycyjne argumentu PanelHandle.
  - Z menu edytora wywołania funkcji wybrać pozycję Code/Select Variable.
  - Zaznaczyć Show Project Variables.
  - Z listy wyświetlonych zmiennych wybrać właściwą. Tutaj należy podwójnie kliknąć na pozycji panelHandle.
  - Nazwa zmiennej zostaje wpisana do pola argumentu.

- Do pola ControlID należy wpisać nazwę stałej przypisanej w tym przypadku do panelu wyświetlacza. Postępuje się identycznie jak poprzednio z tym, że należy wybrać pozycję Code/Select UI Constant. Z wyświetlonej listy stałych należy tutaj wybrać nazwę PANEL_NUMERIC. Wyświetlaczowi podczas projektowania GUI przypisano stałą o nazwie NUMERIC, ale LW automatycznie dodaje przedrostek w postaci nazwy panelu macierzystego. W ten sposób nazwy stałych przypisanych do paneli potomnych wskazują na swój panel macierzysty.
- Do pola Value należy wpisać nazwę zmiennej według, której wyświetlacz numeryczny ma być uaktualniany. Postępuje się identycznie jak przy polu PanelHandle. Należy wybrać zmienną x.
- Pole Status reprezentuje wartość zwracaną przez funkcję. Zwykle jest to kod wykonania funkcji, którego własności są listowane po kliknięciu prawym klawiszem myszki na tym polu. Wartość zwracaną często przypisuje się do określonej zmiennej. Nazwę tej zmiennej można wprowadzić do pola identycznie jak w poprzednie. Kreowana instrukcja przyjmuje wtedy postać wyrażenia przypisania. W przykładzie pole pozostawia się puste, ponieważ nie założono wykorzystania kodu błędu.
- Teraz wystarczy przenieść utworzone wyrażenie do kodu źródłowego aplikacji. W tym celu trzeba z menu edycji wywołania funkcji wybrać operację Code/Insert Function Call i zamknąć okno edycji wywołania funkcji.

Po opisanych operacjach funkcja obsługi przełącznika dwupozycyjnego ma postać taką jak pokazano na rys.5-4. Patrząc wstecz można powiedzieć, że operacje te są zbyt skomplikowane. Jest to z pewnością prawdą dla tak prostego przypadku jak wywołanie funkcji SetCtrlVal. Jeśli jednak program jest złożony, wykorzystuje wiele zmiennych i stosuje się specyficzne funkcje z wielu argumentami wtedy przedstawiony mechanizm edycji wywołania funkcji staje się przydatny.

Działanie aplikacji.

Po przedstawionych modyfikacjach uzyskuje się działającą aplikację (rys.5-6). Każde przestawienie przełącznika inkrementuje zmienną x a jej aktualną zawartość prezentuje wyświetlacz numeryczny.



Rys.5-6. Panel działającej aplikacji.

6. Projektowanie aplikacji w LabWindows; przykład 3.

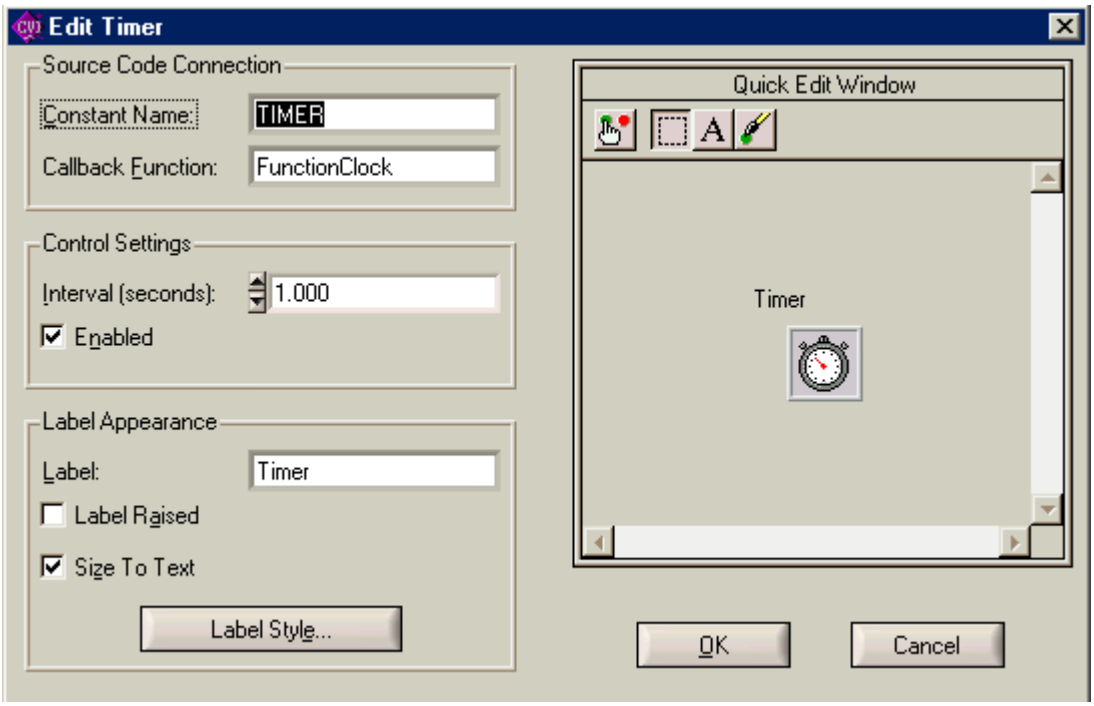
W poprzednich przykładach funkcje callback były wywoływane zdarzeniami produkowanymi użyciem myszki lub klawiatury. Istnieje możliwość wykorzystania zdarzenia zegarowego (EVENT_TIMER_TICK) produkowanego przez zegar programowy LW do realizacji pewnych zadań w odstępach czasu określonych wystąpieniem takiego zdarzenia. W tym celu GUI należy wyposażyć w zegar oraz wyposażyć go w odpowiednią funkcję callback.

Aplikacja ma zliczać zdarzenia czasowe w warunkach ustawienia przełącznika w pozycji ON. Aktualną wartość zmiennej licznikowej prezentuje wyświetlacz numeryczny. Interfejs użytkownika aplikacji musi być rozbudowany w stosunku do przykładu pierwszego o trzy dodatkowe elementy: zegar programowy, element nastawczy w postaci przełącznika binarnego oraz element prezentacyjny w postaci pola numerycznego.

Dodanie zegara programowego, przełącznika i pola numerycznego do GUI aplikacji.

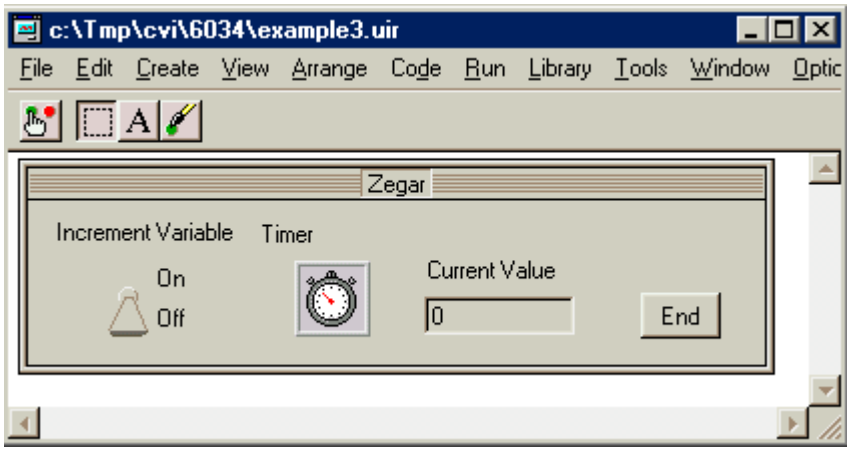
Po uruchomieniu LW otwiera się okno projektu. Należy otworzyć projekt przykładu 1 i przejść do edytora UIR. Po uaktywnieniu panelu macierzystego (kliknięcie w obszarze jego okna), za pomocą pozycji Create/Timer menu edytora wykreować zegar programowy. Teraz należy określić żądane jego własności (rys.6-1). W tym przypadku należy:

- Określić nazwę funkcji callback związanej z przełącznikiem, np. *FunctionClock*.
- Określić przedział czasowy generacji zdarzeń (*Interval*), np. 1sek.
- Etykietę zmienić na postać opisującą jego przeznaczenie, np. *Timer*.



Rys.6-1. Panel edycji własności obiektu zegara programowego.

Pozostałe elementy interfejsu GUI, przełącznik i pole numeryczne, wykreować podobnie jak w przykładzie 2 (rys.6-2).



Rys.6-2.GUI aplikacji.

Dla zegara oraz przełącznika wykreować szkielety funkcji obsługujących ich zdarzenia, które otrzymają postać przedstawioną na rys.6-3 i 6-4.

```
int CVICALLBACK FunctionClock (int panel, int control, int event,
                               void *callbackData, int eventData1, int eventData2)
{
        switch (event)
                {
                   case EVENT_TIMER_TICK:

                        break;
                }
        return 0;
}
```

Rys.6-3. Szkielet funkcji FunctionClock.

```
int CVICALLBACK FunctionIncrement (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
        switch (event)
                {
                 case EVENT_COMMIT:

                        break;
                }
        return 0;
}
```

Rys.6-3. Szkielet funkcji FunctionIncrement.

Aplikacja potrzebuje dwóch zmiennych globalnych (rys.6-4). Jednej stanowiącej licznik zdarzeń zegarowych (x) oraz drugiej przechowującej stan przełącznika (iOnOff).

```
// Dodatkowe zmienne globalne aplikacji
static int iOnOff;          // Stan przełącznika – 1 gdy ON
static int x;               // Licznik zdarzeń
```

Rys.6-4. Zmienne globalne aplikacji.

Dołączony kod do funkcji FunktionClock (rys.6-5) sprawdza stan zmiennej iOnOff i jeśli oznacza on włączenie przełącznika wykonuje inkrementację zmiennej licznikowej oraz wywołuje funkcję uaktualniającą stan wyświetlacza numerycznego.

```
int CVICALLBACK FunctionClock (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
        switch (event)
                {
                case EVENT_TIMER_TICK:
                        if ( iOnOff )
                                {
                                x++;
                                SetCtrlVal(panelHandle, PANEL_NUMERIC, x);
                                }
                        break;
                }
        return 0;
}
```

Rys.6-5. Kompletna funkcja FunctionClock.

Funkcja obsługi zdarzenia przełącznika FunctionIncrement zajmuje się wyłącznie odczytem stanu przełącznika po zdarzeniu EVENT_COMMIT i wpisaniem 0 lub 1 do zmiennej iOnOff, z której korzysta funkcja obsługi zdarzeń zegarowych. Przełącznik mógłby nie posiadać funkcji callback, ponieważ odczyt stanu przełącznika można zrealizować w funkcji obsługi zdarzenia zegarowego. Jednak obsługa zdarzenia zegarowego jest szybsza przy wykorzystaniu zmiennej globalnej iOnOff w porównaniu z wywołaniem funkcji GetCtrlVal.

```
int CVICALLBACK FunctionIncrement (int panel, int control, int event,
                    void *callbackData, int eventData1, int eventData2)
{
        switch (event)
                {
                case EVENT_COMMIT:
                        GetCtrlVal (panelHandle, PANEL_BINARYSWITCH, &iOnOff);
                        break;
                }
        return 0;
}
```

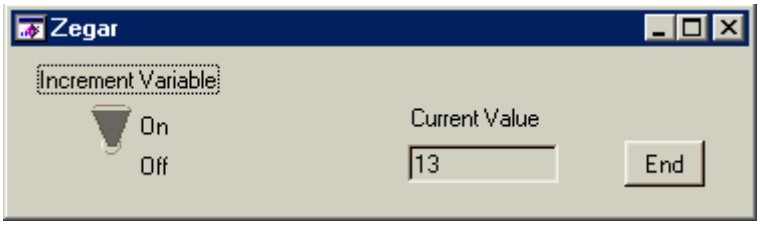
Rys.6-6. Kompletna funkcja FunctionIncrement.

Funkcja obsługi zdarzeń zegarowych musi realizować swoje zadania bardzo sprawnie, tak aby czas jej wykonania był znacznie krótszy od założonych odstępów czasowych wystąpienia zdarzeń zegarowych. W przeciwnym razie nastąpi zmonopolizowanie procesora przez ich obsługę, co w krytycznej sytuacji może uniemożliwić nawet przerwanie działania aplikacji nie mówiąc już o

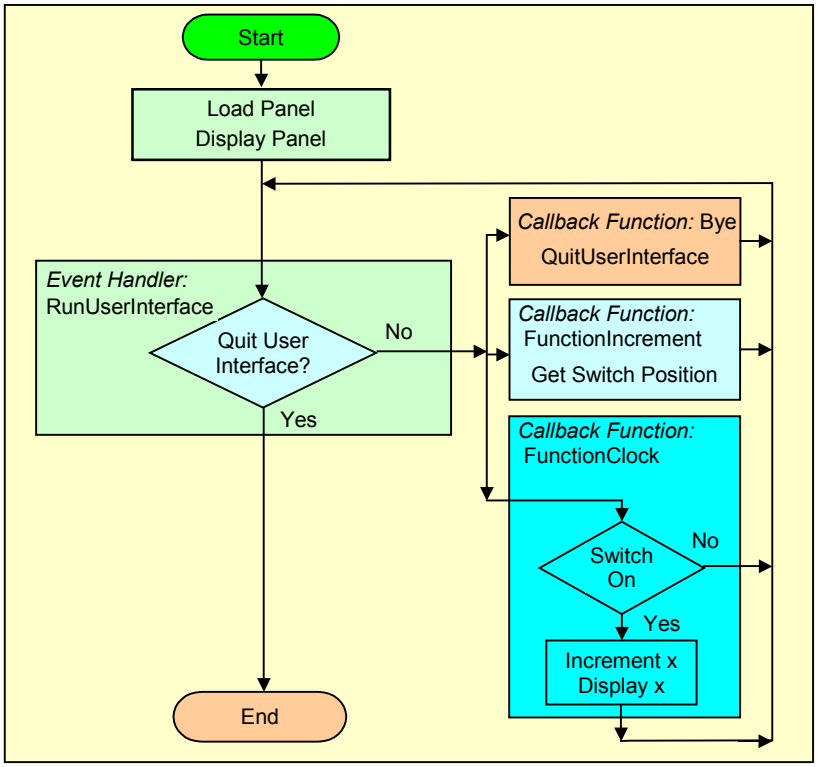
niemożliwości obsługi innych zdarzeń. Jest to szczególnie istotne przy krótkich przedziałach czasu. Dodatkowo dokładność wyznaczenia momentów czasowych zmniejsza się przy krótkich odstępach, ponieważ procesy o wyższym priorytecie blokują przez pewien czas powrót do obsługi procesu korzystającego ze zdarzeń czasowych.

Aplikacja może wykorzystywać zdarzenia czasowe do realizacji różnych zadań. Można to uzyskać przez zastosowanie kilku zegarów odmierzających zadane odstępy czasowe i przypisanie do nich odrębnych funkcji callback realizujących unikalne zadania. W tej sytuacji nie można zapewnić synchronizacji działania zegarów. Jeśli występuje taka konieczność można zastosować jeden zegar, którego funkcja callback zlicza zdarzenia oraz w zależności od stanu licznika realizuje określone zadania, np. po każdym zdarzeniu odczytuje wynik pomiaru z przyrządu a co dziesiąte zdarzenie wyświetla ostatnią partię odczytanych wyników.

Okno działającej aplikacji zliczającej jednosekundowe zdarzenia zegarowe w stanie ON przełącznika przedstawia rys.6-7. Diagram obrazujący jej działanie pokazuje rys.6-8.



Rys.6-7. Okno działającej aplikacji.



Rys.6-8. Diagram aplikacji.

7. Podsumowanie.

Środowisko LW pozwala projektować złożone aplikacje pomiarowe przeznaczone do pracy w systemie operacyjnym Windows. Jest doskonałym narzędziem projektowym również dla osób nie będących profesjonalnymi programistami dzięki zaimplementowaniu specyficznego sposobu

projektowania oraz wyposażenia środowiska w bogate biblioteki dotyczące interfejsu użytkownika, operacji matematycznych, obsługi różnorodnych urządzeń pomiarowych itp. Algorytm postępowania projektowego powinien przedstawiać się następująco:

- Przyjąć ogólną koncepcję rozwiązania, listując wejścia i wyjścia aplikacji. Przedstawić aplikację w postaci diagramu określając wymagane funkcje callback oraz ich przeznaczenie.
- Zaprojektować interfejs GUI. Dla poszczególnych elementów interfejsu określić ich charakter (sterujący/prezentacyjny) oraz dodać nazwy funkcji callback do wszystkich elementów sterujących. Koniecznie należy dodać element sterujący zakończeniem działania aplikacji.
- Wygenerować szkielet kodu programu. Wskazać funkcji QuitUserInterface element sterujący odpowiedzialny za przerwanie działania aplikacji.
- Do każdej funkcji callback dodać kod odpowiedzialny za realizację przewidzianych operacji.
- Wykonać kompilację i linkowanie programu oraz sprawdzić jego działanie. Ten etap obejmuje likwidację błędów oraz dopracowanie szczegółów projektu i najczęściej jest on bardzo pracochłonny.

8. Literatura

[1] Kurt Wick; C & LabWindows Fundamentals; University of Minnesota;
(http://mxp.physics.umn.edu/resources/Documentation.htm)

[2] National Instruments; Windows GUI Techniques for Instrumentation Programmers with LabWindows/CVI; University of Minnesota; Instrupedia 2002; AN054;