



Politechnika Wrocławska

Wydział Elektroniki,  
Fotoniki i Mikrosystemów

---

## Laboratorium informatyki

### Ćwiczenie nr 12. Maszyny stanów

Zagadnienia do opracowania:

- przykłady maszyn stanów
- diagram stanów i tabela przejść między stanami
- metody implementacji maszyn stanów

---

## Spis treści

<b>1</b>	<b>Cel ćwiczenia</b>	<b>2</b>
<b>2</b>	<b>Wprowadzenie</b>	<b>2</b>
2.1	Maszyna stanów . . . . .	2
2.2	Implementacja maszyny stanów z użyciem instrukcji warunkowej <i>switch-case</i> . . . . .	4
2.3	Implementacja maszyny stanów z użyciem wskaźników funkcyjnych . . . . .	8
<b>3</b>	<b>Program ćwiczenia</b>	<b>17</b>
<b>4</b>	<b>Dodatek</b>	<b>19</b>
4.1	Mętne wskaźniki i idiom pImpl . . . . .	19

---

## 1. Cel ćwiczenia

Celem ćwiczenia jest opanowanie umiejętności implementacji maszyn stanów w językach *C/C++* do zastosowań praktycznych.

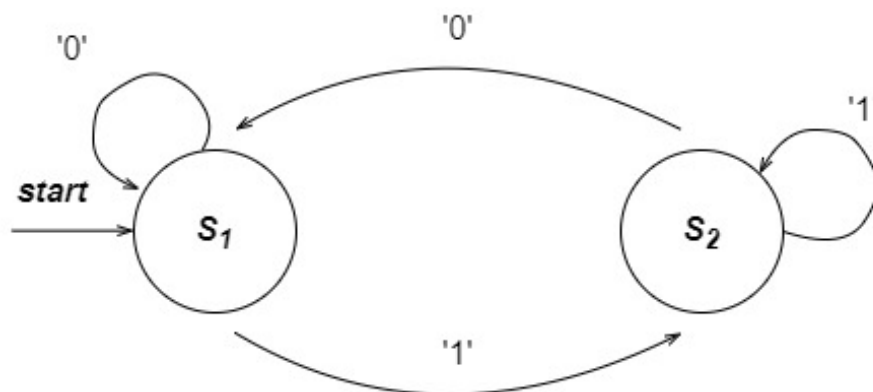
## 2. Wprowadzenie

### 2.1. Maszyna stanów

**Maszyna stanów**, zwana również **automatem skończonym** (*ang. finite-state machine, FSM*), to model matematyczny abstrakcyjnego obiektu, który może przyjmować określone wartości (**stany**). Model określa możliwe dyskretne przejścia (*ang. transitions*) między **stanami**. Obiekt może przebywać tylko w jednym **stanie** w danym momencie. Zmiana **stanu** odbywa się na podstawie sygnałów wejściowych docierających do obiektu. Spośród różnych implementacji automatów można wyróżnić **automat Moore’a** oraz **automat Mealy’ego**. Struktura **automatu Moore’a** zakłada, że **wartość sygnału wyjściowego zależy wyłącznie od stanu wewnętrznego obiektu**. W przypadku **automatu Mealy’ego** **wartość sygnału wyjściowego dodatkowo zależy od aktualnej wartości sygnałów wejściowych**.

Przykładem prostej **maszyny stanów** może być układ mikrokontrolera z diodą elektroluminescencyjną (LED), która w **stanie początkowym** jest wygaszona. Mikrokontroler za pomocą interfejsu szeregowego odbiera ciąg bitów. Jeżeli ostatni odebrany bit jest równy **1** – wystawiany jest stan wysoki na wyprowadzeniu diody (dioda zaczyna świecić), jeżeli bit jest równy **0** – dioda jest wygaszana przez ustawienie stanu niskiego. Przedstawiony układ może zostać zamodelowany za pomocą dwóch **stanów**:  $S_1$  – dioda nie świeci,  $S_2$  – dioda świeci. **Maszynę stanów** można wówczas zobrazować za pomocą **diagramu stanów** (rys. 2.1). Początkowo układ jest w stanie  $S_1$ . Jeżeli kolejno odbierane bity mają wartość **0**, to układ pozostaje w **stanie**  $S_1$ .

Jeżeli odebrano bit **1**, to następuje przejście do **stanu**  $S_2$ . Układ pozostaje w **stanie**  $S_2$  tak długo, jak odbierane bity mają wartość **1**.



Rys. 2.1. Diagram stanów układu mikrokontrolera z diodą elektroluminescencyjną

Zależność między sygnałami wejściowymi a **stanami** układu można przedstawić również w postaci tabeli przejść (tabela 1).

Tabela 1. Tabela przejść maszyny stanów z rys. 2.1

Stan \ Wejście	0	1
$S_1$	$S_1$	$S_2$
$S_2$	$S_1$	$S_2$

---

## 2.2. Implementacja maszyny stanów z użyciem instrukcji warunkowej *switch-case*

Na listingu 1. przedstawiono przykład prostej implementacji maszyny stanów z rys 2.1. **Stany**  $S_1$  i  $S_2$  zostały zdefiniowane za pomocą typu wyliczeniowego **State**:

```
1 typedef enum State {  
2     LED_OFF, // Stan S1  
3     LED_ON  // Stan S2  
4 } State_t;
```

Obsługa diody elektroluminescencyjnej odbywa się z wykorzystaniem funkcji **turnOnLed()** oraz **turnOffLed()**. Kolejne bity przesyłane interfejsem szeregowym są odczytywane za pomocą funkcji **getNextBit()**:

```
1 void turnOnLed();  
2 void turnOffLed();  
3 unsigned short getNextBit();
```

Aktualny **stan maszyny** przechowywany jest w zmiennej **currentState** (początkowo zainicjalizowanej wartością **LED\_OFF**). W pętli pobierane są kolejne bity (**nextBit**), przesłane za pomocą interfejsu szeregowego. Przejścia między **stanem**  $S_1$  (**LED\_OFF**) a **stanem**  $S_2$  (**LED\_ON**) odbywają się wewnątrz instrukcji warunkowej **switch-case**. Mechanizm działania **maszyny** (układu mikrokontrolera) w określonym **stanie** został zawarty w obrębie pojedynczej etykiety **case**. Warto zwrócić uwagę, że pętla z listingu 1. teoretycznie będzie wykonywać się w nieskończoność (*while(true)*). Jest to celowy zabieg w przypadku mikrokontrolerów nieposiadających systemu operacyjnego – program ma się wykonywać tak długo, aż użytkownik nie zatrzyma działania mikrokontrolera i nie umieści w pamięci *Flash* nowego

---

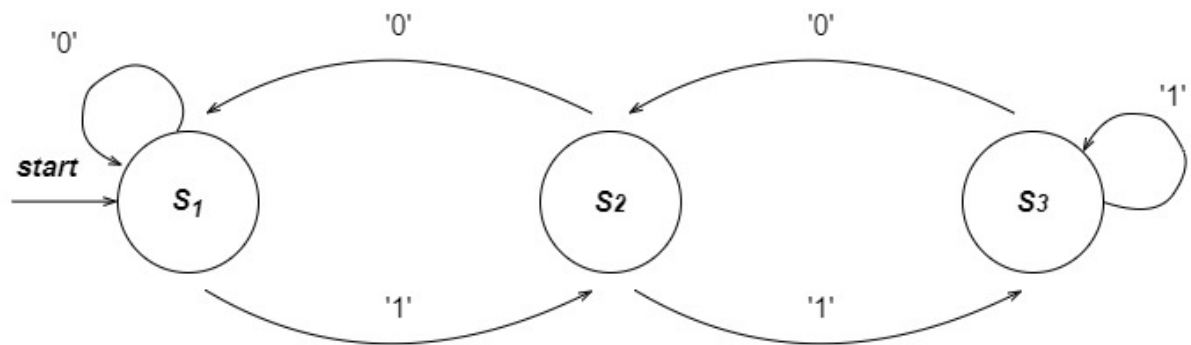
kodu wykonywalnego.

```
1 State_t currentState = LED_OFF;
2 while (true) {
3     unsigned short nextBit = getNextBit();
4     switch (currentState) {
5         case LED_OFF:
6             if (nextBit == 1) {
7                 turnOnLed();
8                 currentState = LED_ON;
9             }
10            break;
11        case LED_ON:
12            if (nextBit == 0) {
13                turnOffLed();
14                currentState = LED_OFF;
15            }
16            break;
17    }
18 }
```

Listing 1. Prosta implementacja maszyny stanów z rys. 2.1

Rozwiązanie z listingu 1., mimo swojej prostoty, jest obarczone poważnymi wadami. Przede wszystkim, w przedstawionym rozwiązaniu, program dopuszcza (teoretycznie) dowolne przejścia między *stanami* obiektu. W przypadku *maszyny* z rys. 2.1. nie ma to żadnego znaczenia (dostępne są tylko dwa *stany*). Sytuacja zaczyna się komplikować dla bardziej rozbudowanych układów. Na rys. 2.2. przedstawiono *diagram stanów* układu mikrokontrolera, do którego dołączono drugą diodę. Świecenie drugiej diody określono jako *stan*  $S_3$ . Mikrokontroler przechodzi w *stan*  $S_3$  tylko, jeżeli wcześniej został wystawiony stan wysoki na wyprowadzeniu diody pierwszej ( $S_2$ ) i ko-

lejnny odebrany bit ma wartość 1. Wygaszanie diod odbywa się w kolejności odwrotnej.



Rys. 2.2. Diagram stanów układu mikrokontrolera z dwiema diodami elektroluminescencyjnymi

Tabela przejść dla rozbudowanej *maszyny stanów* została przedstawiona w tabeli 2. Widać, że nie ma możliwości bezpośredniego przejścia ze stanu  $S_1$  do  $S_3$  i odwrotnie.

Tabela 2. Tabela przejść maszyny stanów z rys. 2.2

Stan \ Wejście	0	1
$S_1$	$S_1$	$S_2$
$S_2$	$S_1$	$S_3$
$S_3$	$S_2$	$S_3$

Implementacja *maszyny stanów* zostałaby w tym przypadku rozszerzona tak, jak przedstawiono na listingu 2. Mimo, że instrukcje warunkowe *if-else* wewnątrz etykiet *case* sterują odpowiednimi przejściami *między stanami*, to sama instrukcja warunkowa *switch-case* w żaden sposób nie zabrania przejścia ze *stanu*  $S_1$  (*LED\_OFF*) do  $S_3$  (*SECOND\_LED\_ON*). Ponadto zauważalnie rośnie rozmiar funkcji implementującej *maszynę* i zmniejsza się jej czytelność.

```
1 State_t currentState = LED_OFF;
2 while (true) {
3     unsigned short nextBit = getNextBit();
4     switch (currentState) {
5         case LED_OFF:
6             if (nextBit == 1) {
7                 turnOnLed();
8                 currentState = LED_ON;
9             }
10            break;
11        case LED_ON:
12            if (nextBit == 0) {
13                turnOffLed();
14                currentState = LED_OFF;
15            } else {
16                turnOnSecondLed();
17                currentState = SECOND_LED_ON;
18            }
19            break;
20        case SECOND_LED_ON:
21            if (nextBit == 0) {
22                turnOffSecondLed();
23                currentState = LED_ON;
24            }
25            break;
26    }
27 }
```

Listing 2. Implementacja maszyny stanów z rys. 2.2

Dalsze rozbudowywanie *maszyny stanów* skutkowałoby powstaniem zagmatwanego, trudnego w utrzymaniu kodu (*ang. spaghetti code*). Nietrud-

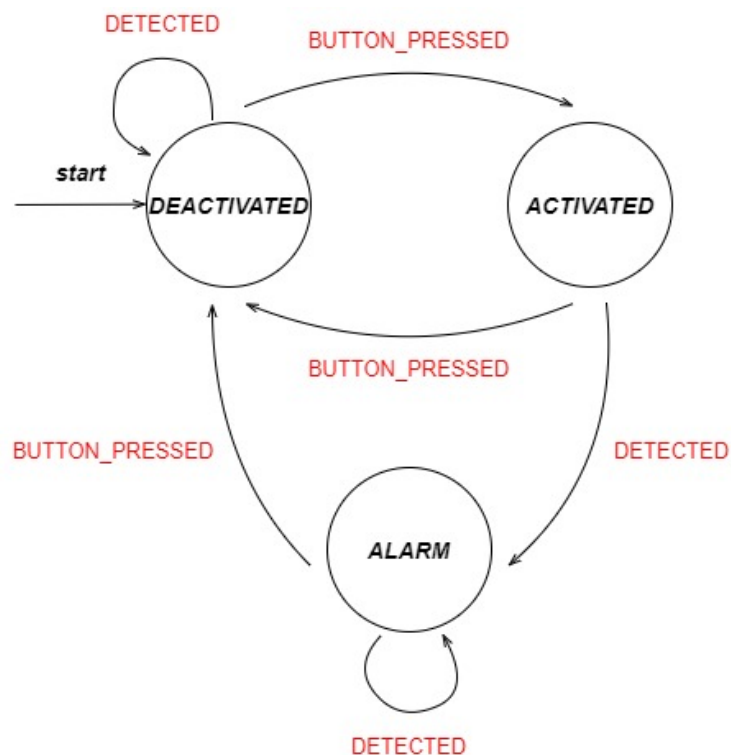


---

no wówczas o pomyłkę. Przez nieuwagę można przypisać do zmiennej *currentState* niewłaściwą wartość, skutkując niedozwolonym przejściem między *stanami* (jak np.  $S_1 \rightarrow S_3$ ). Dlatego implementacja *maszyn stanów* z użyciem instrukcji warunkowej *switch-case* nie jest wskazana w przypadku bardziej rozbudowanych systemów.

## 2.3. Implementacja maszyny stanów z użyciem wskaźników funkcyjnych

Na rys. 2.3. przedstawiono przykład *maszyny stanów* systemu alarmowego. Układ składa się z mikrokontrolera wyposażonego w brzęczyk (*ang. buzzer*), detektor ruchu oraz przycisk, służący do aktywacji bądź dezaktywacji alarmu. System obsługuje dwa *zdarzenia* (które mogą być zrealizowane np. za pomocą *przerwań programowych* (*ang. interrupts*)) – **DETECTED**, pochodzące od detektora ruchu, oraz **BUTTON\_PRESSED**, wywoływane przy wciśnięciu przycisku. Na rys. 2.3. *zdarzenia* te zostały zaznaczone barwą czerwoną. System może znajdować się w jednym z trzech *stanów*: **DEACTIVATED** – alarm rozbrojony, **ACTIVATED** – alarm aktywny, **ALARM** – alarm uruchomiony.



Rys. 2.3. Maszyna stanów systemu alarmowego

W tabeli 3. zestawiono możliwe przejścia między poszczególnymi *stanami*. Mikrokontroler ignoruje *zdarzenie* od detektora ruchu tak długo, jak pozostaje w *stanie DEACTIVATED*. Aktywacja alarmu odbywa się przez wciśnięcie przycisku (*zdarzenie BUTTON\_PRESSED*). Ponowne wciśnięcie przycisku rozbraja alarm. Jeżeli alarm jest uruchomiony (*stan ALARM*), to włączony zostaje brzęczyk, aż do naciśnięcia przycisku (przejście w *stan DEACTIVATED*).

Tabela 3. Tabela przejść maszyny stanów z rys. 2.3

Stan \ Zdarzenie	DETECTED	BUTTON_PRESSED
DEACTIVATED	DEACTIVATED	ACTIVATED
ACTIVATED	ALARM	DEACTIVATED
ALARM	ALARM	DEACTIVATED

---

Ponieważ implementacja wykorzystująca instrukcję warunkową *switch-case* nie jest odpowiednia dla *maszyny stanów* przedstawionej na rys. 2.3, rozwiązaniem może być implementacja wykorzystująca struktury i wskaźniki funkcyjne. Każdy *stan* jest zaimplementowany jako struktura przechowująca deskryptor (identyfikator) *stanu* oraz funkcję obsługi *zdarzeń*. Deskryptory *stanów* i *zdarzeń* mogą zostać wówczas zaimplementowane z wykorzystaniem enumeratorów:

```
1 typedef enum StateDescriptor {  
2     DEACTIVATED ,  
3     ACTIVATED ,  
4     ALARM  
5 } StateDescriptor_t;  
6  
7 typedef enum EventDescriptor {  
8     BUTTON_PRESSED ,  
9     DETECTED  
10 } EventDescriptor_t;
```

Każda funkcja obsługi *zdarzenia* (*handler*), w obrębie pojedynczego *stanu*, jest mapowana na deskryptor *zdarzenia*. Każda z funkcji zwraca deskryptor *stanu*, w który nastąpi przejście po obsłudze danego *zdarzenia*. Takie rozwiązanie pozwala uniknąć kosztownej dynamicznej alokacji pamięci dla całej struktury, przy każdorazowym wywoływaniu funkcji obsługi *zdarzenia*. Strukturę odzwierciedlającą *stan* maszyny przedstawiono na listingu 3.

---

```

1 typedef StateDescriptor_t (*EventHandler)();
2
3 typedef struct State {
4     StateDescriptor_t stateDescriptor;
5     EventDescriptor_t eventDescriptor;
6     EventHandler handler;
7 } State_t;

```

Listing 3. Struktura reprezentująca stan maszyny z rys. 2.3

Funkcje obsługi *zdarzeń* dla poszczególnych stanów przedstawiono na listingu 4. Funkcje *void turnOnBuzzer()* i *void turnOffBuzzer()* realizują obsługę brzęczyka.

```

1 // Obsługa zdarzenia DETECTED w stanie DEACTIVATED
2 StateDescriptor_t idleDeactivated() {
3     return DEACTIVATED;
4 }
5 // Obsługa zdarzenia BUTTON_PRESSED w stanie
6 // DEACTIVATED
7 StateDescriptor_t activate() {
8     return ACTIVATED;
9 }
10 // Obsługa zdarzenia BUTTON_PRESSED w stanie
11 // ACTIVATED / ALARM
12 StateDescriptor_t deactivate() {
13     turnOffBuzzer();
14     return DEACTIVATED;
15 }
16 // Obsługa zdarzenia DETECTED w stanie ACTIVATED
17 StateDescriptor_t raiseAlarm() {
18     turnOnBuzzer();

```

```

17     return ALARM;
18 }
19 // Obsługa zdarzenia DETECTED w stanie ALARM
20 StateDescriptor_t idleAlarm() {
21     return ALARM;
22 }

```

Listing 4. Funkcje obsługi zdarzeń dla poszczególnych stanów

Na listingu 5. przedstawiono funkcję *handleEvent()*, realizującą *maszynę stanów* z rys. 2.3. Funkcja ta wywoływana jest każdorazowo, kiedy wygenerowane zostanie nowe *zdarzenie* (np. w *podprogramie obsługi przerwania*). Tablica *stateMachine* przechowuje zainicjalizowane struktury *stanów DEACTIVATED, ACTIVATED* oraz *ALARM* dla *zdarzeń BUTTON\_PRESSED* i *DETECTED*. *Maszyna stanów* została zdeklarowana z wykorzystaniem słowa kluczowego *static*. Dzięki temu tworzona jest tylko raz w czasie działania aplikacji (optymalizacja czasu wykonania programu). Zmienna *currentState* przechowuje deskryptor aktualnego *stanu maszyny*. W pętli *for* następuje wywołanie odpowiedniej funkcji obsługi *zdarzenia* (*handler*). W wyniku porównania aktualnego deskryptora *stanu maszyny* (*currentState*) oraz deskryptora *zdarzenia* (*event*) z wartościami przypisanymi do kolejnych elementów tablicy *stateMachine* następuje wybór właściwej instancji struktury *State*. Rezultat zwrócony przez funkcję obsługi *zdarzenia* jest zapisywany w zmiennej *currentState*. Po obsłudze *zdarzenia* wywoływana jest instrukcja *break*. Dalsza iteracja nie ma sensu, ponieważ każda para deskryptorów *stanu* i *zdarzenia* w tablicy *stateMachine* jest unikalna.

---

```

1 void handleEvent(EventDescriptor_t event) {
2     static const State_t stateMachine[] = {
3         {DEACTIVATED, BUTTON_PRESSED, activate},
4         {DEACTIVATED, DETECTED, idleDeactivated},
5         {ACTIVATED, BUTTON_PRESSED, deactivate},
6         {ACTIVATED, DETECTED, raiseAlarm},
7         {ALARM, BUTTON_PRESSED, deactivate},
8         {ALARM, DETECTED, idleAlarm}
9     };
10
11     static StateDescriptor_t currentState =
DEACTIVATED;
12     for (unsigned int i = 0; i < sizeof(stateMachine)
) / sizeof(State_t); ++i) {
13         if (stateMachine[i].stateDescriptor ==
currentState && stateMachine[i].eventDescriptor
== event) {
14             currentState = stateMachine[i].handler()
;
15             break;
16         }
17     }
18 }

```

Listing 5. Funkcja realizująca maszynę stanów z rys. 2.3

Dużą zaletą przedstawionej implementacji *maszyny stanów* jest jej **skalowalność**. Aby rozszerzyć *maszynę* o nowy *stan* należy:

- rozszerzyć typ wyliczeniowy *StateDescriptor* o nowy enumerator;
- zdefiniować funkcje obsługi poszczególnych *zdarzeń* dla nowego *stanu*;

- 
- dodać zainicjalizowane struktury *State* do tablicy *stateMachine* dla wszystkich *zdarzeń* obsługiwanych w nowym *stanie*.

Aby dodać nowe *zdarzenie* należy:

- rozszerzyć typ wyliczeniowy *EventDescriptor* o nowy enumerator;
- zdefiniować funkcje obsługi *zdarzenia* dla każdego z istniejących *stanów*;
- dodać zainicjalizowane struktury *State* do tablicy *stateMachine* dla nowego *zdarzenia* obsługiwanego we wszystkich *stanach*.

Przedstawioną strukturę *maszyny stanów* można uprościć, jeżeli odpowiednio zmodyfikuje się wskaźnik funkcyjny *EventHandler* tak, jak na listingu 6. Wówczas typ wyliczeniowy *StateDescriptor* staje się zbędny, a struktura *State* posiada dwa pola, będące wskaźnikami funkcyjnymi: *handler* oraz *executor*. Wskaźnik *handler* przechowuje adres jednej z trzech funkcji odpowiedzialnych za obsługę konkretnego *zdarzenia* i w związku z tym, za przejście do nowego *stanu*. Te funkcje to odpowiednio: *deactivated()*, *activated()* albo *alarm()*. Wskaźnik *executor* przechowuje adres funkcji, która ma zostać wykonana po przejściu do nowego *stanu*. W przypadku *stanu DEACTIVATED* jest to funkcja *turnOffBuzzer()*, a w przypadku *stanu ALARM* funkcja *turnOnBuzzer()*. Dla *stanu ACTIVATED* wprowadzono funkcję *idle()*, której wywołanie nie niesie ze sobą żadnych skutków ubocznych:

```
1 void idle() {}
```

W celu uniknięcia każdorazowej dynamicznej alokacji pamięci dla struktury *State* podczas wykonywania funkcji przypisanej do wskaźnika *handler*, wprowadzono predefiniowane stałe globalne instancje *stanów DEACTIVATED*, *ACTIVATED* oraz *ALARM*. Specyfikator *static* zapewnia wiązanie wewnętrzne – do predefiniowanych instancji *stanów* można odwołać się tylko wewnątrz danej jednostki translacji (por. Ćw. 3, Rodzaje

---

zmiennych). Takie podejście umożliwia również znaczne uproszczenie funkcji *handleEvent()*. Statyczna tablica struktur *State* została zastąpiona statycznym wskaźnikiem na strukturę *State*. Wskaźnik *state* jest jednorazowo inicjalizowany adresem do statycznej stałej *DEACTIVATED* podczas pierwszego wywołania funkcji *handleEvent()*. Wywołanie funkcji przypisanej do wskaźnika *handler* skutkuje przejściem do nowego *stanu* – adres zwrócony z funkcji zostaje zapisany pod zmienną *state* (aktualny *stan maszyny*). W ostatnim kroku wywoływana jest funkcja przypisana do wskaźnika *executor*. W ten sposób zrealizowana została obsługa brzęczyka (funkcje *turnOnBuzzer()* i *turnOffBuzzer()*).

```
1 typedef enum EventDescriptor {
2     BUTTON_PRESSED,
3     DETECTED
4 } EventDescriptor_t;
5
6 // Deklaracje zapowiadające
7 typedef struct State State_t;
8 const State_t * deactivated(EventDescriptor_t event)
9     ;
10 const State_t * activated(EventDescriptor_t event);
11 const State_t * alarm(EventDescriptor_t event);
12
13 typedef const State_t * (*EventHandler)(
14     EventDescriptor_t);
15 typedef void (*StateExecutor)();
16
17 struct State {
18     EventHandler handler;
19     StateExecutor executor;
20 };
21
```



```

19
20 // Predefiniowane stany
21 static const State_t DEACTIVATED = {deactivated,
    turnOffBuzzer};
22 static const State_t ACTIVATED = {activated, idle};
23 static const State_t ALARM = {alarm, turnOnBuzzer};
24
25 // Obsługa zdarzenia w stanie DEACTIVATED
26 const State_t * deactivated(EventDescriptor_t event)
    {
27     return event == BUTTON_PRESSED ? &ACTIVATED : &
        DEACTIVATED;
28 }
29 // Obsługa zdarzenia w stanie ACTIVATED
30 const State_t * activated(EventDescriptor_t event) {
31     return event == BUTTON_PRESSED ? &DEACTIVATED : &
        ALARM;
32 }
33 // Obsługa zdarzenia w stanie ALARM
34 const State_t * alarm(EventDescriptor_t event) {
35     return event == BUTTON_PRESSED ? &DEACTIVATED : &
        ALARM;
36 }
37
38 void handleEvent(EventDescriptor_t event) {
39     static const State_t * state = &DEACTIVATED;
40     state = state->handler(event);
41     state->executor();
42 }

```

Listing 6. Implementacja maszyny stanów z rys. 2.3

---

### 3. Program ćwiczenia

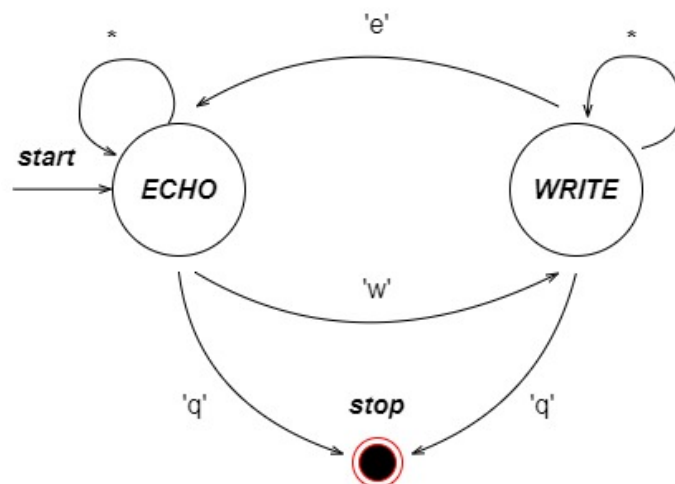
**Zadanie 1.** Celem zadania jest implementacja aplikacji przetwarzającej znaki pobierane z klawiatury. Program działa bazując na dwóch stanach:

- **ECHO** – realizuje funkcję echa (wysyła odebrane znaki na standardowe wyjście);
- **WRITE** – zapisuje odebrane znaki do pliku tekstowego *output.txt*.

Program zaczyna swoje działanie w stanie **ECHO** (rys. 3.1). Przejście ze stanu **ECHO** do stanu **WRITE** odbywa się po odebraniu znaku 'w', natomiast przejście odwrotne po odebraniu znaku 'e'. Po odebraniu znaku 'q' program kończy działanie. W ramach zadania rozpisz tabelę przejść dla maszyny stanów przedstawionej na rys. 3.1 (dowolny znak poza 'e', 'w' i 'q' oznaczono jako '\*'), a następnie, wykorzystując dowolne podejście, zaimplementuj opisaną aplikację. W tym celu:

- w pliku *state.c* zdefiniuj funkcje *echo()* oraz *write()*, obsługujące pobierane znaki odpowiednio w stanie **ECHO** oraz **WRITE**. Funkcje powinny przyjmować obsługiwany znak jako argument wywołania;
- w pliku *statemachine.c* zdefiniuj funkcję *void exec()* realizującą działanie maszyny stanów. Deklaracja funkcji została umieszczona w pliku nagłówkowym *statemachine.h*. Odczyt kolejnych znaków z klawiatury powinien odbywać się z wykorzystaniem funkcji *char readChar()*, której deklarację zamieszczono w pliku nagłówkowym *reader.h*.

Wywołanie maszyny stanów zostało zaimplementowane wewnątrz funkcji *main()* (plik *main.c*).



Rys. 3.1. Maszyna stanów z zadania 1.

**Zadanie 2.** Program pobiera z klawiatury ciąg liczb całkowitych i przeprowadza na nich odpowiednie obliczenia. Jeżeli **aktualnie pobrana** liczba:

- **jest dodatnia** – to **następna** liczba będzie **sumowana** z poprzednio uzyskanym wynikiem;
- **jest ujemna** – to **następna** liczba będzie **mnożona** przez poprzednio uzyskany wynik;
- **jest równa 0** – to program kończy działanie.

Po każdym wykonanym działaniu aktualny wynik jest wyświetlany na ekranie komputera. Początkowa wartość wyniku wynosi **1**. W ramach zadania zaprojektuj maszynę stanów realizującą opisane działanie, to jest **narysuj diagram stanów oraz tabelę przejść między stanami**.

**Zadanie 3.** Wykorzystując wskaźniki funkcyjne zaimplementuj maszynę stanów z **Zadania 2**.

---

## 4. Dodatek

### 4.1. Mętne wskaźniki i idiom pImpl

Języki *C/C++* są językami typowanymi statycznie, co oznacza, że zgodność typów zmiennych jest weryfikowana na etapie kompilacji pliku wykonywalnego. Ponadto zmienne czy funkcje kompilowane są zgodnie z kolejnością występowania ich definicji w obrębie danej jednostki translacji, co w przypadku zależności krzyżowych między poszczególnymi typami zmiennych wymusza stosowanie deklaracji zapowiadających. Przykład przedstawiono na listingu 7.

```
1 // Deklaracja zapowiadająca
2 typedef struct State State_t;
3
4 // Typ EventHandler korzysta z typu State_t
5 typedef const State_t * (*EventHandler)(
6     EventDescriptor_t);
7
8 // Krzyżowa zależność State_t - EventHandler
9 struct State {
10     EventHandler handler;
11     StateExecutor executor;
12 };
13
```

Listing 7. Krzyżowe zależności między typami zmiennych (z listingu 6.)

Próba utworzenia instancji (obiektu) struktury, dla której dostępna jest wyłącznie deklaracja zapowiadająca zakończy się błędem kompilacji z informacją o niekompletnym typie zmiennej. Kompilator nie ma informacji ile pamięci musiałby statycznie zaalokować na tworzoną zmienną. Dozwolone jest natomiast definiowanie wskaźników i referencji do niekompletnego typu

---

zmiennej (listing 8). Rozmiar wskaźnika jest zależny od architektury procesora, a nie od typu wskazywanej zmiennej.

```
1 // Deklaracja zapowiadająca
2 typedef struct State State_t;
3
4 // OK - referencja do niekompletnego typu
5 void func(State_t & state) {
6     ...
7 }
8
9 // OK - wskaźnik na niekompletny typ
10 State_t * statePtr;
11
12 // Błąd kompilacji - niekompletny typ State_t
13 State_t someState;
```

Listing 8. Wskaźniki i referencje do niekompletnego typu zmiennej

W językach *C/C++* wskaźniki i referencje do niekompletnego typu zmiennej, których przykłady przedstawiono na listingu 8., zwykle się nazywać *meta-nymi wskaźnikami (referencjami)* (*ang. opaque pointers (references)*). Są one stosowane w celu:

- optymalizacji czasu kompilacji;
- ukrycia szczegółów implementacyjnych przed użytkownikiem-klientem;
- rozdzielenia warstwy abstrakcji od warstwy implementacji (por. Strukturalny wzorzec projektowy **Most** (*ang. Bridge*)).

Na listingu 9. przedstawiono zawartość pliku nagłówkowego *Student.h* zawierającego definicje struktury **Student\_t** oraz trzech funkcji operujących na zmiennej strukturalnej. Struktura *Student.h* składa się z trzech pól: dwóch

---

tablic znaków (**university** – nazwa uczelni, **department** – nazwa wydziału) oraz wskaźnika do struktury typu **PersonalData\_t** (**pImpl**). Plik nagłówkowy zawiera wyłącznie deklarację zapowiadającą struktury **PersonalData\_t**, zatem wskaźnik **pImpl** jest *mętnym wskaźnikiem*.

```
1 #pragma once
2
3 #include <stdint.h>
4
5 #define MAX_UNIVERSITY_NAME_LENGTH (64)
6 #define MAX_DEPARTMENT_NAME_LENGTH (64)
7
8 // Deklaracja zapowiadająca struktury PersonalData_t
9 typedef struct PersonalData PersonalData_t;
10
11 typedef struct Student {
12     char university[MAX_UNIVERSITY_NAME_LENGTH + 1];
13     char department[MAX_DEPARTMENT_NAME_LENGTH + 1];
14
15     // Wskaznik do implementacji
16     PersonalData_t * pImpl;
17 } Student_t;
18
19 void initialize(Student_t *, const char * university
20               , const char * department);
21
22 void release(Student_t *);
23
24 void introduce(const Student_t const *);
```

Listing 9. Idiom pImpl; plik nagłówkowy

---

Na listingu 10. zamieszczono zawartość pliku źródłowego *Student.c*. Znajdują się w nim definicje struktury **PersonalData\_t** oraz funkcji **initialize()**, **release()** i **introduce()**. Funkcja **initialize()** ustawia wartości wszystkich pól zmiennej strukturalnej typu **Student\_t**, w szczególności dynamicznie alokuje pamięć dla pola **pImpl** i przypisuje predefiniowane wartości do pól **name**, **surname** oraz **index** struktury **PersonalData\_t**. Jest to możliwe, ponieważ plik źródłowy dostarcza definicję struktury **PersonalData\_t** (jej rozmiar jest znany już na etapie kompilacji). Funkcja **release()** zwalnia dynamicznie zaalokowaną pamięć na atrybut **pImpl** zmiennej typu **Student\_t**, natomiast funkcja **introduce()** wypisuje na standardowe wyjście wartości wszystkich pól przekazanej zmiennej strukturalnej.

```
1 #include <stdbool.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 #include "Student.h"
7
8 #define MAX_NAME_LENGTH (16)
9 #define MAX_SURNAME_LENGTH (32)
10
11 // Definicja struktury PersonalData_t
12 struct PersonalData {
13     char name[MAX_NAME_LENGTH + 1];
14     char surname[MAX_SURNAME_LENGTH + 1];
15     uint32_t index;
16 };
17
18 // Statyczna funkcja pomocnicza
```

```
19 static bool checkPointer(const Student_t const *
    student) {
20     bool result = (student != NULL);
21     if (!result) {
22         printf("Provided student does not exist\n");
23     }
24     return result;
25 }
26
27
28 void initialize(Student_t * student, const char *
    university, const char * department) {
29     if (checkPointer(student)) {
30         size_t universityNameLength = strlen(university)
31         ;
32         if (universityNameLength >
33             MAX_UNIVERSITY_NAME_LENGTH) {
34             printf("University name too long. Names up to
35             %d characters are supported\n",
36             MAX_UNIVERSITY_NAME_LENGTH);
37         } else {
38             strncpy(student->university, university,
39             universityNameLength);
40         }
41
42         size_t departmentNameLength = strlen(department)
43         ;
44         if (departmentNameLength >
45             MAX_UNIVERSITY_NAME_LENGTH) {
```



```

39     printf("Department name too long. Names up to
%d characters are supported\n",
MAX_DEPARTMENT_NAME_LENGTH);
40 } else {
41     strncpy(student->department, department,
departmentNameLength);
42 }
43
44 // Predefiniowane wartosci pol name, surname i
index struktury PersonalData_t
45 static const char * name = "Adam";
46 static const char * surname = "Nowak";
47 static const uint32_t index = 123456;
48 // Dynamiczna alokacja pamieci na strukture
PersonalData_t
49 student->pImpl = malloc(sizeof(*student->pImpl))
;
50 // Odwołanie do prywatnych atrybutow struktury
Student_t
51 strncpy(student->pImpl->name, name, strlen(name)
);
52 strncpy(student->pImpl->surname, surname, strlen
(surname));
53 student->pImpl->index = index;
54 }
55 }
56
57 void release(Student_t * student) {
58     if (checkPointer(student)) {
59         // Zwolnienie dynamicznie zaalokowanej pamieci
60         free(student->pImpl);

```

---

```

61     }
62 }
63
64 void introduce(const Student_t const * student) {
65     if (checkPointer(student)) {
66         printf("This is %s %s (index: %d) from %s, %s
67             .\n",
68             student->pImpl->name,
69             student->pImpl->surname,
70             student->pImpl->index,
71             student->university,
72             student->department);
73     }
74 }

```

Listing 10. Idiom pImpl; plik źródłowy

Przedstawione rozwiązanie umożliwia niezależną pracę nad interfejsem struktury **Student\_t** (plik nagłówkowy *Student.h*) oraz jej implementacją (plik źródłowy *Student.c*). Interfejs może być rozszerzany o kolejne funkcje, niezależnie od zawartości struktury **PersonalData\_t**. Co więcej, przeniesienie definicji struktury **PersonalData\_t** z pliku nagłówkowego do pliku źródłowego umożliwia skrócenie czasu kompilacji pliku wykonywalnego aplikacji. Dyrektywa preprocesora **#include** wkleja zawartość importowanego pliku nagłówkowego do każdego z wywołujących ją plików źródłowych. Jeżeli plik nagłówkowy zostanie zmodyfikowany, to kompilator przeprowadzi ponowną kompilację każdej z jednostek translacji odwołujących się do tego pliku. W rozwiązaniu z listingów 9. i 10. zmiana definicji (implementacji) struktury **PersonalData\_t** wymusi ponowną kompilację wyłącznie jednego pliku – *Student.c*, ponieważ plik nagłówkowy zawiera wyłącznie deklarację zapowiadającą struktury. Utworzenie i zwolnienie instancji struktury **Student\_t** może być przeprowadzone następująco:

---

```
1 #include "Student.h"
2
3 int main() {
4     Student_t student;
5     initialize(&student, "Politechnika Wroclawska", "
6         Wydzial Elektryczny");
7     introduce(&student);
8     release(&student);
9     return 0;
}
```

Rozwiązanie w którym deklaracja struktury zawiera *mętny wskaźnik* do swojej implementacji jest nazywana idiomem *pImpl* (*ang. pointer to implementation*) i jest stosowana szczególnie często w *bibliotekach programistycznych* (więcej na temat bibliotek programistycznych w Ćw. 13). Takie podejście może mieć na celu również ukrycie szczegółów implementacyjnych przed użytkownikiem (np. implementacja biblioteki kryptograficznej), który otrzymuje skompilowany plik biblioteki oraz pakiet plików nagłówkowych. W rozwiązaniu z listingów 9. i 10. struktura **PersonalData\_t** zawiera dane wrażliwe studenta, a jej zawartość (implementacja) może być modyfikowana (przez programistę) w razie potrzeb w obrębie pliku źródłowego. Zmiana definicji struktury **PersonalData\_t** wpłynie na wynik działania funkcji **introduce()**, natomiast w żaden sposób nie wpłynie na sposób jej obsługi (wywołania) przez użytkownika.

Stosowanie idiomu *pImpl* może prowadzić do zwiększenia narzutu pamięciowego związanego z dynamiczną alokacją i zwalnianiem pamięci (patrz: funkcje **initialize()** i **release()** z listingu 10.), a także do zmniejszenia czytelności kodu i wzrostu jego skomplikowania. Tak jak każdy wzorzec projektowy, tak i idiom *pImpl* powinien być stosowany świadomie, po rozpatrzeniu potencjalnych zysków i strat wynikających z jego implementacji.

---

**Zadanie (dla chętnych)** Zaimplementuj maszynę stanów z **Zadania 3.**  
z wykorzystaniem idiomu **pImpl**.