Realizacja bloków funkcjonalnych IEC 61131-3 za pomocą maszyn stanowych PIO

Marcin Hubacz, Bartosz Trybus, Bartosz Pawłowicz
Politechnika Rzeszowska, Wydział Elektrotechniki i Informatyki, ul. Wincentego Pola 2, 35-959 Rzeszów

Streszczenie: W pracy zaproponowano hybrydową architekturę oprogramowania sterownika, w której główny algorytm sterowania jest realizowany w językach IEC 61131-3 przez maszynę wirtualną, ale ze wsparciem bloków natywnych wykorzystujących maszyny stanowe PIO (Programmable Input/Output) dostępnych w układach RP2040. Analizowano różne implementacje bloków RS, SR i CTD z użyciem PIO, porównując czas ich realizacji z implementacją w języku ST. Wykazano, że bloki natywne PIO są wielokrotnie szybsze niż te wykonywane przez maszynę wirtualną. Mimo pewnych ograniczeń, takich jak konieczność wcześniejszego zaprogramowania bloków w firmware czy ograniczona liczba instrukcji w blokach PIO, rozwiązanie to oferuje istotne korzyści w aplikacjach, gdzie czas wykonywania operacji jest kluczowy.

Słowa kluczowe: IEC 61131-3, PIO State Machine, blok funkcjonalny, sterownik PLC

1. Wprowadzenie

Współczesne sterowniki programowalne typu PLC (ang. *Programmable Logic Controllers*) oraz PAC (ang. *Programmable Automation Controllers*) charakteryzują się rosnącą złożonością. Oprócz realizacji podstawowej funkcji, jaką jest wykonywanie algorytmów sterowania, urządzenia te muszą często mieć wbudowane zaawansowane mechanizmy komunikacyjne, obejmujące obsługę różnorodnych łączy i protokołów sieciowych [1, 2], a także zapewniać rozbudowaną obsługę sygnałów wejściowych i wyjściowych. Standard IEC (PN-EN) 61131-3 [3] narzuca obowiązek obsługi kilku języków programowania (ST, LD, FBD, IL oraz SFC), jak również wymóg wymiany danych z innymi urządzeniami, będącymi elementami rozproszonego systemu sterowania.

Realizacja tych zadań wymaga od sterownika funkcjonowania w trybie wielozadaniowym, co oznacza konieczność jednoczesnego wykonywania wielu operacji. Wielozadaniowość można osiągnąć przez zastosowanie wielordzeniowej jednostki centralnej [4]. W przypadku sterowników dwurdzeniowych jeden rdzeń może być przeznaczony do obsługi procesów sterowania w czasie rzeczywistym, a drugi do obsługi komunikacji. Możliwa jest także realizacja procesów sterowania o różnych cyklach czasowych [5]. Jeśli liczba rdzeni okazuje się niewystarczająca, konieczne jest użycie systemów operacyjnych, takich jak FreeRTOS czy Linux, które pozwalają na dzielenie czasu procesora między zadania. W konsekwencji struktura oprogramowania sterownika (tzw. *firmware*) staje się wysoce skomplikowana, a wymagania sprzętowe wzrastają, co może stanowić istotne ograniczenie dla niewielkich i budżetowych urządzeń sterujących.

Różnorodność architektur sprzętowych, w tym procesorów centralnych, wymaga odpowiedniego przygotowania algorytmów sterujących napisanych w językach zgodnych z normą IEC 61131-3 przed ich przesłaniem. Klasycznym rozwiązaniem jest kompilacja programów sterujących do kodu maszynowego dedykowanego procesora. Takie podejście utrudnia przenoszenie oprogramowania na sprzęt heterogeniczny, co znacząco ogranicza elastyczność systemu. Alternatywą jest zastosowanie maszyn wirtualnych (VM), które interpretują kod sterujący niezależnie od platformy sprzętowej [6]. Rozwiązanie to uniezależnia oprogramowanie od sprzętu, jednak odbywa się to kosztem wydajności.

Kompromisem poprawiającym wydajność może być rozwiązanie, w którym programista opracowuje zasadniczy przebieg algorytmu sterowania dla maszyny wirtualnej, korzystając przy tym z wcześniej przygotowanych, zoptymalizowanych dla danej platformy sprzętowej bloków funkcjonalnych. Standard IEC 61131-3 obejmuje przy tym zestaw podstawowych bloków funkcjonalnych, takich jak liczniki, detektory zbocza, czasomierze czy przerzutniki. Ich implementacja wewnętrzna może być zależna od platformy sprzętowej i nie wymaga interpretacji przez maszynę wirtualną. Konieczna jest jednak możliwość wywoływania tych bloków przez kod nadrzędny, np. w języku ST, FBD czy LD. W środowisku CPDev, opartym na maszynie wirtualnej, bloki takie nazywane są blokami natywnymi. Służą one zazwyczaj do udostępniania funkcjonalności sprzętowych platformy i są implementowane w języku C. Istnieją również przykłady ich implementacji w języku Verilog dla platform FPGA [7].

W ostatnim czasie pojawiła się interesująca możliwość implementacji bloków natywnych z wykorzystaniem mechanizmu programowalnych wejść/wyjść PIO (ang. *Programmable Input/Output*) [9] w popularnych i ekonomicznych mikrokontrolerach RP2040/2350. W niniejszej pracy zaproponowano architekturę

Autor korespondujący:
Marcin Hubacz, m.hubacz@prz.edu.pl

Artykuł recenzowany
nadesłany 30.11.2024 r., przyjęty do druku 16.05.2025 r.

oprogramowania sterownika wykorzystującego tę koncepcję, co może znacząco przyspieszyć realizację wybranych bloków funkcjonalnych IEC 61131-3. Przedstawiono przykłady implementacji kilku bloków funkcjonalnych (przerzutników i liczników) oraz przeprowadzono badania mające na celu ocenę wzrostu wydajności w porównaniu do rozwiązań opartych wyłącznie na maszynie wirtualnej.

2. Mechanizm PIO State Machine

Wielordzeniowość jest ważną cechą współczesnych urządzeń sterujących, umożliwiającą zwiększenie wydajności i efektywności w przetwarzaniu współbieżnych zadań. Jednak procesory wyposażone w dwa, cztery lub więcej rdzeni często okazują się nadmiarowe w przypadku prostszych zastosowań, takich jak obsługa wejść i wyjść (I/O), realizacja przerzutników czy sekwencyjne sterowanie urządzeniami. Funkcjonowanie takich systemów zazwyczaj wymaga zastosowania wielozadaniowego systemu operacyjnego, często opartego na Linuksie, co wprowadza istotny narzut związany z obsługą I/O. Skutkiem tego jest wydłużenie czasu przetwarzania danych, co w aplikacjach krytycznych może uniemożliwić spełnienie wymagań czasu rzeczywistego.

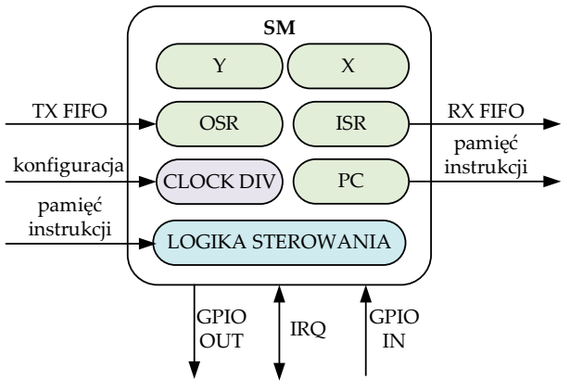
Mikrokontrolery oferują bardziej elastyczne i efektywne podejście do obsługi I/O, dodatkowo dedykowane rozwiązania sprzętowe zapewniają najbardziej wydajną i precyzyjną obsługę sygnałów w czasie rzeczywistym, jednocześnie odciążając główny procesor. Przykładami takich technologii są omawiany w niniejszej pracy mechanizm PIO oraz FlexIO dla układów NXP [10]. Umożliwiają one redukcję złożoności wynikającej z konieczności stosowania wielordzeniowych procesorów i wielozadaniowych systemów operacyjnych, przy jednoczesnym zachowaniu deterministycznego przetwarzania, które jest niezbędne w aplikacjach wymagających wysokiej precyzji.

Mechanizm PIO umożliwia programiście definiowanie operacji związanych z obsługą i sterowaniem pinami GPIO (ang. *General Purpose Input/Output*) w formie instrukcji przypominających kod asemblera. Instrukcje te są wykonywane przez sprzętowe maszyny stanowe SM (ang. *State Machines*), działające niezależnie od głównego programu. Mechanizm PIO jest przydatny szczególnie w zadaniach wymagających deterministycznej obsługi sygnałów i minimalizacji opóźnień. Zastosowania obejmują m.in. generowanie sygnałów o wysokiej precyzji oraz implementację protokołów komunikacyjnych, które nie mają sprzętowego wsparcia w układzie.

W układach RP2040 dostępne są dwa moduły PIO, z których każdy zawiera cztery niezależne maszyny stanowe (razem 8 SM), zaś w RP2350 są trzy takie moduły (12 SM). PIO działa równolegle do głównego procesora o architekturze ARM Cortex-M0+, co pozwala na odciążenie go i realizację krytycznych czasowo zadań I/O. Każda maszyna stanowa jest wyposażona w zestaw rejestrów i zasobów (rys. 1):
- rejestry przesuwne: ISR (ang. *Input Shift Register*), OSR (ang. *Output Shift Register*),
- pamięci tymczasowe: X i Y,
- licznik programu: PC (ang. *Program Counter*),
- kolejki FIFO: TX i RX.

Dzięki nim możliwe jest nie tylko efektywne operowanie w obrębie samego modułu PIO, ale także płynna wymiana danych między CPU a PIO.

Język asemblera PIO składa się z kilku podstawowych operacji, definiowanych za pomocą trzech najstarszych bitów kodu instrukcji. Operacje te obejmują:
- Przesunięcia i transfery danych: IN, OUT, MOV – umożliwiają manipulowanie danymi oraz ich przesyłanie między rejestrami i pinami GPIO.



Rys. 1. Uproszczona struktura maszyny stanowej w PIO
Fig. 1. Simplified state machine structure in PIO

- Sterowanie przepływem programu: JMP, WAIT – pozwalają na bezpośrednią kontrolę przepływu programu za pomocą skoków lub oczekiwania na spełnienie warunków.
- Manipulacje czasem: NOP – wprowadza opóźnienie w cyklu wykonywania.
- Obsługę przerwań: IRQ – umożliwia generowanie i obsługę sygnałów przerwań.
- Przenoszenie danych między kolejkami FIFO a rejestrami przesuwnymi: PUSH, PULL – służą do efektywnej wymiany informacji między CPU a PIO.

Każda instrukcja asemblera PIO jest wykonywana w jednym cyklu zegarowym, co zapewnia deterministyczne działanie w aplikacjach czasu rzeczywistego. Dodatkowo, każda instrukcja może zawierać bity ustawienia bocznego (ang. *side-set*) oraz dodatkowe opóźnienie (ang. *delay*). Mechanizm *side-set* pozwala na równoległą manipulację bitami na pinach GPIO niezależnie od głównej operacji.

Mimo licznych zalet, moduły PIO mają pewne istotne ograniczenia, które mogą wpływać na ich zastosowanie w bardziej złożonych projektach. Przede wszystkim każdy moduł PIO zawiera jedynie cztery maszyny stanowe, co ogranicza liczbę protokołów lub sygnałów, które można obsługiwać równocześnie. Wymusza to precyzyjne planowanie zasobów i często prowadzi do konieczności kompromisów projektowych. Dodatkowym ograniczeniem jest długość programu dla modułu PIO, która wynosi maksymalnie 32 instrukcji. Także zredukowany zestaw instrukcji może wymagać stosowania złożonych, nienaturalnych rozwiązań programistycznych, które komplikują implementację bardziej zaawansowanych funkcji.

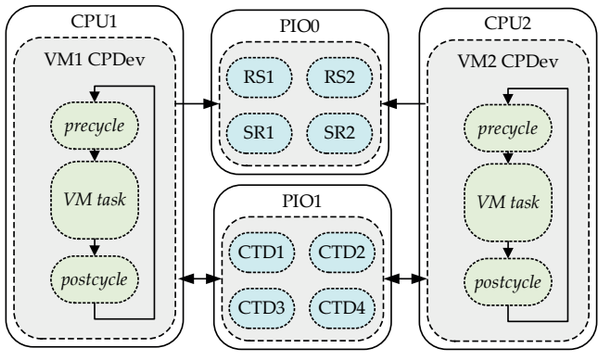
Moduł PIO taktowany jest konfigurowalnym sygnałem zegarowym o domyślnej częstotliwości 125 MHz. Dzielnik zegara (CLOCK DIV na rys. 2) pozwala zmniejszyć tę częstotliwość do 7,5 Hz [9]. Jednakże ograniczeniem jest fakt, że dzielnik, a więc i sygnał taktujący jest współdzielony przez wszystkie maszyny stanowe w danej instancji modułu PIO. Może to powodować trudności w uzyskaniu wymaganej precyzji czasowej i elastyczności. Problem ten jest szczególnie widoczny w aplikacjach, które wymagają generowania wielokanałowych sygnałów PWM lub implementacji niestandardowych protokołów komunikacyjnych o złożonych sekwencjach czasowych.

3. Architektura sterownika z PIO

Sterowniki PLC mogą być wyposażone w rozszerzenia i peryferia umożliwiające obsługę specjalistycznych protokołów komunikacyjnych i magistral, co znacznie zwiększa ich funkcjonalność. Standaryzowana wymiana informacji między głównym programem sterującym a zewnętrznymi modułami pozwala na integrację szerokiej gamy rozwiązań.

Omawiane tutaj podejście wykorzystuje maszyny stanowe PIO jako dedykowany podsystem do obsługi sygnałów w czasie rzeczywistym. Dzięki PIO możliwe jest precyzyjne i deterministyczne zarządzanie sygnałami wejściowymi i wyjściowymi, co odciąża główny procesor sterownika. W prezentowanej architekturze PIO przejmuje czasowo krytyczne zadania, takie jak generowanie sygnałów o wysokiej częstotliwości, przetwarzanie protokołów niestandardowych czy realizacja funkcji liczników i przerzutników.

Na rysunku 2 przedstawiono ogólną strukturę dwurdzeniowego sterownika programowalnego opartego na układzie RP2040. Rdzenie CPU1 i CPU2, za pośrednictwem maszyn wirtualnych VM1 i VM2, niezależnie wykonują dwa zadania sterujące zaprogramowane w językach IEC. Czas cyklu każdego zadania sterującego może być ustalany indywidualnie. Mechanizm PIO wspiera te zadania, umożliwiając implementację wybranych bloków normy IEC, które są wykonywane poza głównym cyklem obliczeniowym każdego z rdzeni. W omawianej architekturze założono wymianę informacji między natywnymi blokami PIO, a programami sterującymi za pomocą zmiennych zdefiniowanych w zadaniach sterujących.



Rys. 2. Ogólna struktura dwurdzeniowego sterownika z blokami natywnymi PIO
Fig. 2. General structure of a dual-core controller with native PIO blocks

Z punktu widzenia systemu sterowania istotne jest rozróżnienie sposobów działania maszyn stanowych, które można podzielić na działanie cykliczne oraz wyzwalane na żądanie. Mechanizm PIO, z natury swojego działania, wymusza cykliczne wykonywanie instrukcji zapisanych w pamięci. W sytuacjach wymagających synchronizacji możliwe jest jednak tymczasowe blokowanie lub restartowanie maszyn stanowych. Dzięki temu możliwe są dwa tryby pracy:

— Ciągła praca niezależna od PLC – maszyny stanowe uruchamiane są podczas startu programu i działają autonomicznie, niezależnie od głównego programu sterownika.
— Praca harmonogramowana – maszyny stanowe są wywoływane w zaplanowanych momentach zgodnie z harmonogramem. Pozwala to synchronizować ich działanie z innymi zadaniami sterownika, skracając czas wykonywania cyklu sterującego.

Takie podejście zapewnia dużą elastyczność i efektywność w implementacji zadań sterujących wymagających deterministycznego działania i precyzyjnej kontroli czasowej.

4. Implementacja wybranych bloków

W ramach prowadzonych badań porównano dotychczasową implementację wybranych bloków z biblioteki IEC 61131-3 w języku ST z nową implementacją wykorzystującą mechanizm PIO. Do analiz wybrano bloki przerzutników typu RS i SR (ang. *flip-flop*) oraz licznik typu CTD (ang. *down counter*). Kod w języku ST dla przerzutnika RS został przedstawiony na listingu 1. Blok ten zawiera dwie zmienne wejściowe VAR_INPUT: S i R (SET – ustawiające, RESET – zerujące), obie typu BOOL. Zmienna wyjściowa Q jest obliczana na podstawie bieżących wartości wejść oraz poprzedniego stanu Q.

```
FUNCTION_BLOCK RS
VAR_INPUT
    S : BOOL;
    R : BOOL;
END_VAR
VAR_OUTPUT
    Q : BOOL;
END_VAR
Q := NOT R AND (Q OR S);
END_FUNCTION_BLOCK
```

Listing 1. Kod przerzutnika RS w języku ST
Listing 1. RS Flip-Flop code in ST language

W przypadku przerzutnika SR wyrażenie obliczające wyjście ma postać:

```
Q := S OR (NOT R AND Q);
```

Kompilator środowiska CPDev generuje z kodu źródłowego ST plik binarny zawierający instrukcje dla maszyny wirtualnej. Program wykonywany przez maszynę wirtualną jest interpretowany, co oznacza, że każda instrukcja maszyny wirtualnej jest przekształcana i realizowana za pomocą dostępnych mechanizmów platformy sprzętowej. W praktyce realizacja pojedynczej instrukcji maszyny wirtualnej wymaga co najmniej kilkunastu, a często znacznie większej liczby instrukcji procesora. W związku z tym implementacja oparta na mechanizmie PIO może znacząco zwiększyć szybkość wykonywania bloków. Taka wersja implementacji pozwala na realizację operacji w sposób bardziej wydajny, dzięki odciążeniu głównego procesora przez dedykowane jednostki sprzętowe.

Na listingu 2 przedstawiono przykładowe implementacje bloków RS (po lewej) oraz SR (po prawej) z wykorzystaniem mechanizmu PIO dla układu RP2040.

```
.program rs
    set pins, 0
.wrap_target
begin:
    in pins, 2
    mov x isr
    jmp !x begin
    in null, 31
    mov x isr
    jmp !x reset
    set pins 1
    jmp begin
reset:
    set pins 0
.wrap
```

```
.program sr
    set pins, 0
.wrap_target
begin:
    jmp pin setOut
    in pins 1
    mov x isr
    jmp !x begin
    set pins 0
    jmp begin
setOut:
    set pins 1
.wrap
```

Listing 2. Implementacja PIO przerzutników RS (po lewej) i SR (po prawej)
Listing 2. PIO Implementation of flip-flops: RS (left) and SR (right)

Programy rozpoczynają się od przypisania wartości domyślnej na wyjściu bloku funkcyjnego Q za pomocą instrukcji `set pins 0`. Następnie cyklicznie wykonywana jest część kodu umieszczona między etykietami `.wrap_target` i `.wrap`. Taki układ pozwala zaoszczędzić cykl procesora w przeciwieństwie do alternatywnego korzystania z instrukcji `jmp begin`.

Blok RS (po lewej) na początku cyklu odczytuje dwa kolejne wejścia GPIO będące odpowiednikami RESET i SET. Za pomocą instrukcji `in pins 2` stan wejść trafia do rejestru ISR. Ze względu na ograniczone operandy instrukcji skoku `jmp`, stan wejść musi zostać uprzednio przeniesiony (`mov`) z ISR do X. Jeżeli obydwa wejścia mają stan niski, wyjście pozostaje bez zmian i wykonywany jest skok do etykiety `begin`. Gdy którykolwiek bit wejścia ma stan wysoki, dalsze instrukcje identyfikują, czy jest to sygnał RESET czy SET. Za pomocą przesunięcia bitowego w prawo (ang. *shift*) z wartością 31 z rejestru ISR wyłuskiwany jest najstarszy bit sygnału RESET i ponownie kopiowany do rejestru X. W zależności od wyniku, wyjście jest zerowane lub ustawiane realizując logikę przerzutnika RS.

Blok SR (po prawej) zaimplementowano z wykorzystaniem instrukcji warunkowego skoku `jmp` na podstawie sygnału na określonym pinie. Blok SM ma możliwość skorelowania jednego pinu z taką instrukcją. Tutaj sprawdzane jest, czy na wejściu SET bloku SR jest stan wysoki. Następuje wówczas skok do etykiety *setOut*, gdzie wystawiany jest stan wysoki na wyjściu Q. W przeciwnej sytuacji program odczytuje wejście RESET i na jego podstawie dokonywana jest dalsza realizacja funkcjonalności bloku SR podobnie jak w bloku RS. Zastosowanie w przykładowej implementacji bloku SR skoku warunkowego pozwoliło wyeliminować przesunięcie bitowe.

Implementacja licznika CTD w języku ST została zaprezentowana na listingu 3. Blok ma dwa wejścia typu BOOL – CD (wyzwalające zbocze narastające) oraz LD (ładujące wartość początkową) – oraz jedno wejście PV typu INT, reprezentujące wartość początkową licznika. Działanie bloku polega na zmniejszaniu wartości licznika po wykryciu zbocza narastającego na wejściu CD. Jeśli na wejściu LD pojawi się stan wysoki, licznik zostaje załadowany wartością początkową z wejścia PV. Blok udostępnia dwa wyjścia: Q – ustawiane, gdy licznik osiąga wartość zero, CV – zwracające bieżącą wartość licznika.

```
FUNCTION_BLOCK CTD
VAR_INPUT
    CD : BOOL;
    LD : BOOL;
    PV : INT;
END_VAR
VAR_OUTPUT
    Q : BOOL;
    CV : INT;
END_VAR
VAR
    CDp : BOOL:=FALSE;
END_VAR
IF LD THEN
    CV := PV;
ELSE
    IF (CD AND NOT CDp) THEN
        IF (CV > 0) THEN
            CV := CV - 1;
        END_IF
    END_IF
END_IF
Q := (CV <= 0);
CDp := CD;
END_FUNCTION_BLOCK
```

Listing 3. Kod licznika CTD w języku ST
Listing 3. CTD Counter Code in ST Language

Implementację licznika CTD z wykorzystaniem mechanizmu PIO przedstawiono na listingu 4. W odróżnieniu od implementacji przerzutników zastosowano w niej operacje na kolejkach FIFO (rys. 1). Użycie FIFO umożliwia efektywne przekazywanie danych między maszyną wirtualną a modułem PIO. Wartość wejściowa PV jest przesyłana z maszyny wirtualnej do PIO za pomocą kolejki TX FIFO, zaś bieżąca wartość licznika CV jest przekazywana z modułu PIO do maszyny wirtualnej za pomocą kolejki RX FIFO, co pozwala na bieżący odczyt stanu licznika przez program sterujący.

```
.program ctd
    set pins, 0
.wrap_target
begin:
    in pins, 2
    mov x isr
    jmp !x begin

    in null, 31
    mov x isr

    jmp !x ctd
    jmp load

load:
    pull
    mov y osr
    set pins, 0
    jmp send

ctd:
    jmp !y setOut
    jmp y-- send
    jmp !y setOut
setOut:
    set pins, 1
send:
    mov isr y
    push

    in pins, 1
    mov x isr
    jmp !x begin
    jmp send
.wrap
```

Listing 4. Implementacja PIO licznika CTD
Listing 4. PIO Counter CTD Implementation

Działanie licznika CTD rozpoczyna się od ustawienia wartości domyślnej wyjścia, podobnie jak miało to miejsce w blokach RS i SR. Kod podzielony został na kilka sekcji: ogólną, ładującą, dekrementującą, wystawiająca i wysyłającą. Pierwsza sekcja, rozpoczynająca się etykietą `begin`, sprawdza stan na wejściach LD i CD. Jeżeli oba mają stan niski, następuje zapętlenie do etykiety `begin`. Sekcja `load` inicjalizująca wartość startową bloku jest uruchamiana, gdy wejście LD ma stan wysoki. Za pomocą `pull` wartość PV jest ładowana z kolejki TX FIFO do rejestru OSR, a następnie przenoszona jest do rejestru tymczasowego Y reprezentującego aktualną wartość licznika CV. Sekcja `ctd` dekrementuje dodatnie wartości Y oraz sprawdza, czy licznik został wyzerowany, w celu załączenia wyjścia Q w sekcji `setOut`. Na koniec sekcja `send` przekazuje instrukcją `push` wartość CV z rejestru Y do kolejki wyjściowej RX FIFO i czeka na pojawienie się stanu niskiego na wejściu CD.

Bloki standardu IEC 61131-3 zrealizowane za pomocą mechanizmu PIO powinny być w pełni dostępne dla programistów korzystających z języków zawartych w normie, takich jak ST, FBD czy LD. Oznacza to możliwość deklaracji oraz wywołania takich bloków w ramach programów tworzonych w tych językach. Środowisko CPDev udostępnia w tym celu interfejs `INativeBlock`, który pozwala na definiowanie bloków natywnych wykonywanych niezależnie od maszyny wirtualnej. Implementacja bloku w oparciu o ten interfejs wymaga zaimplementowania dwóch podstawowych funkcji:

1. `InitNativeBlock` – odpowiedzialnej za inicjalizację bloku, w tym konfigurację niezbędnych zasobów sprzętowych i parametrów działania.

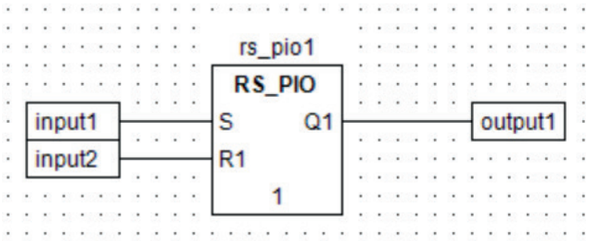
```
class RSPioBlock : public INativeBlock {
    PIO pio;  uint sm, ofs, IO_Pin;

    int InitNativeBlock(WM_ADDRESS32 fb_instance) {
        pio = pio0
        sm = pio_claim_unused_sm(pio, true);
        offset = pio_add_program(pio, &rs_program);
        IO_Pin = fb_instance&0xff;

        gpio_init(IO_Pin);      gpio_set_dir(IO_Pin, GPIO_IN);
        gpio_init(IO_Pin+1);    gpio_set_dir(IO_Pin+1, GPIO_IN);
        gpio_init(IO_Pin+2);    gpio_set_dir(IO_Pin+2, GPIO_OUT);

        rs_program_init(pio, sm, ofs, IO_Pin, IO_Pin+1, IO_Pin+2);
        return 0;
    };

    int ExecNativeBlock(WM_ADDRESS32 fb_instance) {
        pio_sm_set_enabled(pio, sm, true);
        return 0;   // block handled
    };

    void rs_program_init(PIO pio, uint sm, uint offset, uint in_s,
       uint in_r, uint out_q) {
        pio_sm_set_consecutive_pindirs(pio, sm, in_s, 2, false);
        pio_gpio_init(pio, in_s);
        pio_gpio_init(pio, in_r);
        pio_sm_set_consecutive_pindirs(pio, sm, out_q, 1, true);
        pio_gpio_init(pio, out_q);

        pio_sm_config c = rs_program_get_default_config(offset);
        sm_config_set_in_pins(&c, in_s);
        sm_config_set_set_pins(&c, out_q, 1);
        sm_config_set_out_pins(&c, out_q, 1);
        sm_config_set_jmp_pin(&c, in_r);
        pio_sm_init(pio, sm, offset, &c);
    }
};
```

Listing 5. Kod inicjujący blok RS w PIO
Listing 5. Initialization Code for RS Block in PIO

2. `ExecNativeBlock` – realizującej właściwe działanie bloku podczas jego wywołania.

Przykład implementacji bloku RS z użyciem interfejsu `INativeBlock` został przedstawiony na listingu 5. W metodzie `InitNativeBlock` inicjującej blok RS deklarowane są zmienne pomocnicze zawierające wybraną instancję PIO, dostępną (nieprzydzieloną) SM oraz początek instrukcji ładowanego programu asemblerowego. Mechanizm PIO wymusza stosowanie kolejnych numerów pinów GPIO dla wejść i wyjść, czyli niedostępne jest wybranie nieprzypadających po sobie pinów. Tutaj na podstawie najmłodszego bajtu argumentu `fb_instance` przydzielany jest numer `IO_Pin` pierwszego z trzech pinów używanych przez blok RS. Dalej są one inicjowane wraz z odpowiednim kierunkiem pracy (2 wejścia, 1 wyjście). Zadaniem pomocniczej funkcji `rs_program_init` jest przypisanie odpowiedniej konfiguracji dla PIO i SM. W pierwszych jej liniach następuje zainicjowanie w PIO pinów wejściowych i wyjściowych. Następnie ładowana jest domyślna konfiguracja SM z offsetem pierwszej instrukcji programu. Piny wejściowe S i R są ustawiane do odczytu danych i ich przesyłania do ISR. Wyjście Q jest łączone z instrukcjami `set` i `out`, co umożliwia sterowanie jego stanem przez OSR. Ostatnia linia ładuje przygotowaną konfigurację. Za uruchomienie SM odpowiada metoda `ExecNativeBlock`, która wywołuje funkcję `pio_sm_set_enabled` dla określonego PIO i SM.Deklaracja bloku RS_PIO w języku ST przedstawiona na Listingu 6 (po lewej stronie) zawiera znacznik `HARDWARE_BODY_CALL`,

```
FUNCTION_BLOCK RS1_PIO
(*$HARDWARE_BODY_CALL Id:0x0112*)
VAR_INPUT
    S : BOOL;
    R1 : BOOL;
END_VAR
VAR_OUTPUT
    Q1 : BOOL;
END_VAR
END_FUNCTION_BLOCK
```



Listing 6. Definicja bloku natywnego RS PIO w CPDev oraz przykład użycia w języku FBD
Listing 6. Definition of the Native RS PIO Block in CPDev and Example Usage in FBD

który wskazuje, że implementacja bloku jest realizowana poza maszyną wirtualną. Parametr `Id` służy do wyboru instancji bloku RS PIO oraz pierwszego pinu dla tego bloku. Zastosowanie znacznika umożliwia integrację bloku z programami użytkownika w różnych językach IEC 61131-3, takich jak ST, FBD, czy LD. Przykład takiego zastosowania pokazano na Listingu 6 (po prawej stronie), gdzie blok RS_PIO został użyty w diagramie w języku FBD.

5. Rezultaty

Zaproponowane implementacje bloków funkcjonalnych RS, SR i CTD sprawdzono pod kątem czasu wykonywania przez sterownik. Porównano implementację w języku ST wykonywaną przez maszynę wirtualną z realizacją w formie bloków natywnych PIO. W każdym z przypadków zadanie sterujące zawierało wyłącznie wywołanie badanego bloku. Częstotliwość taktowania CPU i PIO ustalono na poziomie 125 MHz czyli maksymalną dla układu RP2040. Maszyna wirtualna uruchomiona została w uniwersalnym 32-bitowym trybie z funkcjami kopiowania pamięci [8].

Do pomiarów czasu rozpoczęcia i zakończenia cyklu sterowania wykonywanego przez maszynę wirtualną zastosowano analizator logiczny. W tabeli 1 przedstawiono wyniki pomiarów, które wskazują czas wykonywania programu przez maszynę wirtualną. Czas trwania zadania może zmieniać się w zależności od struktury kodu programu ST, w szczególności jego przebiegu logicznego i obecności warunków.

Warto dodać, że do pełnego czasu cyklu sterownika należy doliczyć dodatkowe 1–2 µs, które są potrzebne na wykonanie operacji takich jak odczyt wejść, zapis wyjść oraz aktualizację zmiennych w fazach *precycle* i *postcycle* (rys. 2).

Minimalny czas wykonywania bloków implementowanych w PIO jest iloczynem liczby instrukcji wykonywanych programów i odwrotności częstotliwości zegara systemowego:

$$t_{SM} = \frac{1}{f_{clk}} \cdot \sum_{i=1}^{n} I_i$$

W tabeli 2 zestawiono minimalne i maksymalne czasy wykonania bloków opartych na maszynach stanowych PIO. Sekwencja bloku RS trwa 3, 7 lub 8 instrukcji, natomiast blok SR, wykorzystujący skok warunkowy na podstawie sygnału z GPIO, zajmuje 2, 4 lub 6 instrukcji. Implementacja CTD jest bardziej złożona i w pojedynczym cyklu obliczeniowym wymaga od 3 do 15 instrukcji, pomijając zapętlenie oczekujące na zmianę stanu sygnału wejściowego CD. Minimalny czas wykonania każdej instrukcji, zgodnie z zaleceniami producenta, wynosi 8 ns ($f_{clk} = 125$ MHz).

Tabela 1. Czas wykonywania bloków RS, SR, CTD przez maszynę wirtualną
Table 1. Execution time of RS, SR, and CTD blocks by the virtual machine

Blok	Czas min. (µs)	Czas maks. (µs)
RS	49,54	74,43
SR	51,97	75,28
CTD	62,91	88,11

Tabela 2. Czas wykonywania bloków RS, SR, CTD z wykorzystaniem PIO
Table 2. Execution time of RS, SR, and CTD blocks using PIO

Blok	Czas min. (ns)	Czas maks. (ns)
RS	24	48
SR	16	64
CTD	24	120

Jak wynika z danych przedstawionych w tabelach 1 i 2, czas realizacji tych samych bloków funkcjonalnych przez maszynę wirtualną i mechanizm PIO różni się znacząco. Oznacza to, że programista, wykorzystując natywne bloki PIO, może znacząco zwiększyć szybkość wykonywania kodu. W zastosowaniach, w których występują ograniczenia czasowe, może to przynieść istotne korzyści.

6. Podsumowanie

Proponowana architektura oprogramowania sterownika ma formę hybrydową co oznacza, że ogólny algorytm sterowania można dowolnie kształtować i modyfikować za pomocą konstrukcji języków normy IEC 61131-3, jednocześnie wspierając go wyjątkowo szybkimi blokami natywnymi PIO. Dzięki temu możliwe jest znaczące zmniejszenie narzutu czasowego generowanego przez maszynę wirtualną. Jak pokazano, wykonywanie bloków w PIO jest wielokrotnie szybsze niż interpretowanie takich bloków przez maszynę wirtualną.

Rozwiązanie to wiąże się jednak z pewnymi ograniczeniami. Bloki PIO muszą być zaprogramowane wcześniej w firmware

sterownika, co ogranicza elastyczność w tworzeniu oprogramowania zgodnie z normą IEC. Liczba dostępnych bloków PIO i maszyn stanowych w układzie jest również ograniczona. Zestaw instrukcji programów PIO nie może przekraczać 32, co czyni ten mechanizm bardziej odpowiednim dla prostszych operacji, takich jak wiązanie wejść z wyjściami. Innym wyzwaniem jest trudność w debugowaniu kodu PIO.

Mimo tych ograniczeń, bloki natywne PIO stanowią wartościowe narzędzie do przyspieszania algorytmów sterowania, szczególnie w zadaniach o charakterze logicznym.

Bibliografia

1. Wu X., Xie L., *Performance evaluation of industrial Ethernet protocols for networked control application*, "Control Engineering Practice", Vol. 84, 2018, 208–217, DOI: 10.1016/j.conengprac.2018.11.022.
2. Wu X., Xie L., Lim F., *Network delay analysis of EtherCAT and PROFINET IRT protocols*, IECON 2014 the 40th Annual Conference of the IEEE Industrial Electronics Society, 2014. DOI: 10.1109/iecon.2014.7048872.
3. John K.H., Tiegelkamp M., *IEC 61131-3: Programming Industrial Automation Systems*, 2010, DOI: 10.1007/978-3-642-12015-2.
4. Becker M., Sandström K., Behnam M., Nolte T., *A Many-Core Based Execution Framework for IEC 61131-3*, Proceedings of the IECON 2015 – 41st Annual Conference of the IEEE Industrial Electronics Society, Yokohama, Japan, 9–12 November 2015, DOI: 10.1109/IECON.2015.7392805.
5. Hubacz M., Trybus B., *Dual-Core PLC for Cooperating Projects with Software Implementation*, "Electronics", Vol. 12, No. 23, 2023, DOI: 10.3390/electronics12234730.
6. Sadolewski J., Trybus B., *Compiler and virtual machine of a multiplatform control environment*, "Bulletin of the Polish Academy of Sciences Technical Sciences", Vol. 70, No. 2, 2022, DOI: 10.24425/bpasts.2022.140554.
7. Hajduk Z., Sadolewski J., Trybus B., *Multiple tasks in FPGA-based programmable controller*, "e-Informatica Software Engineering Journal", Vol. 5, Nr 1, 2011, 77–85, DOI: 10.2478/v10233-011-0032-2.
8. Hubacz M., Trybus B., *Data Alignment on Embedded CPUs for Programmable Control Devices*, "Electronics", Vol. 11, No. 14, 2022, DOI: 10.3390/electronics11142174.

Inne źródła

9. [https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf] – RP2040 Datasheet, Chapter 3: PIO.
10. [https://www.nxp.com/docs/en/supporting-information/FTF-ACC-F1179_Introduction_to_FlexIO.pdf] – Introduction to FLEXIO.

Implementation of IEC 61131-3 Functional Blocks Using PIO State Machines

Abstract: The paper proposes a hybrid controller architecture where the main control algorithm is executed in IEC 61131-3 languages via a virtual machine, but with support from native blocks using PIO (Programmable Input/Output) state machines available in systems such as the RP2040 is proposed. Various implementations of RS, SR, and CTD blocks using PIO were analyzed, comparing their execution time with that of an implementation in the ST language. It was demonstrated that native PIO blocks are significantly faster than those executed by the virtual machine. Despite some limitations, such as the need to pre-program blocks in firmware or the limited number of instructions in PIO blocks, this solution offers significant benefits in applications where the execution time of operations is critical.

Keywords: IEC 61131-3, PIO State Machine, function block, PLC controller

mgr inż. Marcin Hubacz

m.hubacz@prz.edu.pl
ORCID: 0000-0002-2748-11454

W 2019 r. ukończył studia na Wydziale Elektrotechniki i Informatyki Politechniki Rzeszowskiej – kierunek Automatyka i Robotyka oraz Informatyka. Obecnie Asystent w Katedrze Informatyki i Automatyki Politechniki Rzeszowskiej. Jego główne zainteresowania dotyczą robotyki, elektroniki, systemów wbudowanych oraz druku 3D.

dr inż. Bartosz Pawłowicz

barpaw@prz.edu.pl
ORCID: 0000-0001-9469-2754

Adiunkt w Katedrze Systemów Elektronicznych i Telekomunikacyjnych Wydziału Elektrotechniki i Informatyki Politechniki Rzeszowskiej. Doktorat w dyscyplinie telekomunikacja uzyskał na Wydziale Elektrotechniki, Automatyki, Informatyki i Elektroniki AGH w Krakowie w 2012 r. Jego główne badania dotyczą systemów identyfikacji bezstykowej RFID i ich zastosowań.

dr inż. Bartosz Trybus

btrybus@kia.prz.edu.pl
ORCID: 0000-0002-4588-3973

Adiunkt w Katedrze Informatyki i Automatyki Politechniki Rzeszowskiej. Ukończył studia na Wydziale Elektrycznym, Automatyki, Informatyki i Elektroniki AGH w Krakowie. Doktorat z informatyki uzyskał w 2004 r. Jego główne badania dotyczą systemów czasu rzeczywistego i środowisk wykonawczych oprogramowania sterującego.