POLITECHNIKA RZESZOWSKA
im. IGNACEGO ŁUKASIEWICZA

WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

DZIEDZINA NAUK INŻYNIERYJNO-TECHNICZNYCH

DYSCYPLINA INFORMATYKA TECHNICZNA
I TELEKOMUNIKACJA

PRACA DOKTORSKA

Wielordzeniowa maszyna wirtualna dla sterowników programowalnych

Autor: mgr inż. Marcin Hubacz

Promotor: prof. dr hab. inż. Leszek Trybus
Promotor pomocniczy: dr inż. Bartosz Trybus

Rzeszów, 2026

RZESZOW UNIVERSITY
OF TECHNOLOGY

THE FACULTY OF
ELECTRICAL
AND COMPUTER ENGINEERING
RZESZOW UNIVERSITY OF TECHNOLOGY

FIELD OF SCIENCE ENGINEERING AND TECHNOLOGY

SCIENTIFIC DISCIPLINE COMPUTER SCIENCE AND
TELECOMMUNICATIONS ENGINEERING

DOCTORAL THESIS

Multi-core virtual machine for programmable controllers

Author: Marcin Hubacz

Supervisor: Prof. Leszek Trybus, DSc, Eng.
Assistant Supervisor: Bartosz Trybus, PhD, Eng.

Rzeszow, 2026

Serdeczne podziękowania kieruję do Promotora, Pana prof. dr. hab. inż. Leszka Trybusa, za nieskończone pokłady cierpliwości, wyrozumiałości i nakierowanie na właściwe tory. Mojemu Promotorowi Pomocniczemu, Panu dr. inż. Bartoszowi Trybusowi, dziękuję za bycie niezawodną pierwszą linią wsparcia i falochronem dla moich pomysłów. Dziękuję również Panu dr. inż. Janowi Sadolewskiemu za wsparcie merytoryczne i wiele cennych wskazówek. Na końcu – choć to najważniejsze – dziękuję mojej Żonie Karolinie oraz Synom: Maksymilianowi i Julianowi. Bez Waszych uśmiechów ta praca nie miałaby sensu.

Streszczenie

Przedstawiono metodę rozszerzenia maszyny wirtualnej przeznaczonej dla środowiska jednordzeniowego na środowisko wielordzeniowe, w którym rdzenie procesora wykonują projekty sterowania normy IEC 61131-3 wymieniając między sobą dane. Jako maszynę wirtualną należy rozumieć procesor zrealizowany programowo wykonujący pewien uniwersalny kod pośredni generowany przez kompilator programu źródłowego. Programowa realizacja i uniwersalny kod pośredni umożliwiają implementację maszyny na różnych procesorach. Przykładem jednordzeniowego środowiska z maszyną wirtualną jest CPDev opracowany w PRz i stosowany w sterownikach z procesorami 16- i 32-bitowymi oraz x86-64. CPDev z unikalnym językiem pośrednim VMASM stanowi bazę niniejszej pracy.

Przyjęto, że w każdym rdzeniu procesora funkcjonuje osobna maszyna wirtualna CPDev rozszerzona o wymianę danych z innymi rdzeniami za pośrednictwem pamięci współdzielonej. W związku z tym na metodę rozszerzenia środowiska jednordzeniowego na więcej rdzeni składają się następujące zasady: 1) jednakowe deklaracje wymienianych zmiennych globalnych w projektach dla rdzeni, 2) dodatkowe atrybuty tych zmiennych określające zapis lub odczyt do/z pamięci współdzielonej, 3) wymiana zmiennych na początku i końcu cyklu sterowania wykonywanego przez rdzeń, 4) eliminacja konfliktów rdzeni w dostępie do pamięci współdzielonej. Istotne jest, że nie wymaga się wykorzystania systemu operacyjnego czasu rzeczywistego RTOS.

Przedstawiono model denotacyjny funkcjonowania maszyny wirtualnej, modele i realizacje C/C++ wybranych instrukcji języka pośredniego VMASM, jak również model bezkolizyjnego dostępu do pamięci współdzielonej. Modele denotacyjne wskazują wprost na rozwiązania programowe, które można implementować w dowolnym języku. Pokazano ponadto jak na wydajność maszyny oraz na rozmiar pamięci kodu wpływają trzy metody dostępu do pamięci, tzn. dostęp bajtowy, systemowe *memcpy* oraz bezpośredni dostęp wskaźnikowy.

Pierwszy z przedstawionych przykładów dotyczy wykonywania dwóch powiązanych projektów IEC 61131-3 przez dwa rdzenie mikrokontrolera STM32 na płycie uruchomieniowej. Projekty napisano w językach FBD i ST, a do eliminacji konfliktów w dostępie do pamięci współdzielonej wykorzystano semafor sprzętowy. Ze względu na odmienne cykle wykonywania projektów, do symulacji współpracy rdzeni wykorzystano wielowątkowy WinController.

Drugim przykładem jest laboratoryjny system identyfikacji znaczników RFID, w skład którego wchodzi mały robot mobilny z dwurdzeniowym mikrokontrolerem ESP32 oraz

czterordzeniowy mikrokomputer Raspberry Pi jako jednostka nadrzędna. W systemie funkcjonują trzy maszyny wirtualne przypisane do izolowanych rdzeni, które wymieniają dane poprzez pamięć współdzieloną lub przez Wi-Fi. Mechanizm muteks nie dopuszcza do kolizji w dostępie do pamięci. Systemy operacyjne Linux (Raspberry Pi) i FreeRTOS (ESP32) zostały wykluczone z harmonogramowania zadań, do których przypisano maszyny wirtualne. Eksploracja jest półautomatyczna ze względu na brak bezprzewodowej lokalizacji. Po symulacji wspólnego projektu dla całego systemu rozdzielono go na zadania dla rdzeni.

Abstract

A method for extension of a virtual machine dedicated for single-core environment on multi-core environment is presented. The cores execute control projects of the IEC 61131-3 standard involving mutual exchange of data. The virtual machine is understood as a processor implemented by software which executes certain universal intermediate code generated by a compiler of source program. Implementation by software and universal intermediate code enable installation of the virtual machine on various processors. CPDev developed in RUT is an example of single-core environment involving the virtual machine, and used by industrial controllers with 16- and 32-bit microprocessors, as well as by x86-64. The CPDev with unique VMASM intermediate language is the basis for this work.

It is assumed that each core of the processor involves the CPDev virtual machine extended by data exchange with the other cores through shared memory. Therefore the method for extension of the single-core environment on several cores consists of the following rules: 1) the same declarations of the exchanged global variables in projects for the cores, 2) additional attributes of these variables to define reading or writing from/to the shared memory, 3) exchange of the variables at the beginning and end of the core control cycle, 4) elimination of the core conflicts in the shared memory access. It is important that usage of a RTOS real-time operating system is not required.

Denotational model of the virtual machine operation is presented, as well as the models and C/C++ realizations of selected instructions of the VMASM intermediate language, together with the model of conflict-free access to the shared memory. The denotational models can be implemented in any programming language. In addition, it is shown how efficiency of the virtual machine and size of the code memory depend on memory access methods such as byte access, system *memcpy*, and pointer-based direct access.

The first of the presented examples concerns execution of two related IEC 61131-3 projects by two cores of the STM32 microcontroller in a development board. The projects are written in FBD and ST languages, with conflicts in the shared memory access eliminated by a hardware semaphore. Due to different execution cycles of the projects, the multi-threaded WinController is applied for simulation of the cooperating cores.

The laboratory system for identification of RFID transponders is the second example. The system consists of a small mobile robot with dual-core ESP32 microcontroller and quad-core Raspberry Pi microcomputer as a master unit. Three virtual machines assigned to isolated cores operate in the system, exchanging the data through the shared memory or Wi-Fi. Mutex mechanism prevents memory access conflicts. Linux (Raspberry Pi) and FreeRTOS (ESP32)

operating systems are excluded from scheduling the tasks to which the virtual machines are assigned. After simulation of the common project for the whole system, it has been partitioned into tasks for the cores.

Spis treści

1. Wprowadzenie

Wśród środowisk programistyczno-wykonawczych sterowników znajdują się środowiska bazujące na maszynach wirtualnych. Zaletą ich jest wieloplatformowość, tzn. możliwość wykonywania tego samego kodu binarnego przez różne procesory. Powszechnie stosowanymi środowiskami ogólnego przeznaczenia opartymi na maszynach wirtualnych są Java i .NET. Koncepcję tą wykorzystuje również środowisko CPDev będące bazą dla niniejszej pracy.

Maszyna wirtualna zastosowana w CPDev wykonuje cyklicznie jedno zadanie reprezentujące np. sterownik PLC, albo regulator PID, ale nie obydwa takie zadania jednocześnie. Tymczasem wobec rozpowszechnienia się w ostatnich latach procesorów wielordzeniowych powstała już taka możliwość poprzez umieszczenie zadań w osobnych rdzeniach, z wymianą danych między nimi za pomocą pamięci współdzielonej. Dlatego celem niniejszej pracy jest opracowanie metody rozszerzającej środowisko jednozadaniowe, w tym szczególnie maszynę wirtualną, na procesory wielordzeniowe. Aplikacja metody zarówno w typowym mikrokontrolerze dwurdzeniowym jak i w bardziej złożonym systemie laboratoryjnym zawierającym dwa komunikujące się urządzenia powinna potwierdzić jej użyteczność dla zwiększenia funkcjonalności zwłaszcza małych sterowników, które dotychczas pracują w trybie jednozadaniowym.

1.1. Środowiska programistyczno-wykonawcze sterowników

Zadaniem urządzenia komputerowego pracującego w czasie rzeczywistym jest terminowe reagowanie na zmiany stanu wejść poprzez obliczanie wartości wyjść według zaprogramowanego algorytmu. Na odczyt wejść, obliczenia i aktualizację wyjść jest przeznaczony określony czas, zwany czasem cyklu. Urządzenia funkcjonujące w ten sposób nazywane są ogólnie sterownikami. Wyróżnia się wśród nich następujące grupy:

- sterowniki programowalne PLC (*Programmable Logic Controller*),
- stacje procesowe systemów DCS (*Distributed Control System*),
- przemysłowe komputery IPC (*Industrial PC*),
- sterowniki wbudowane i dedykowane.

Sterowniki PLC są najpowszechniej spotykanym rodzajem urządzeń pracujących w czasie rzeczywistym. W sterownikach wbudowanych i dedykowanych program jest na stałe wpisany do pamięci.

Programowanie sterowników odbywa się zgodnie z standardem IEC 61131-3 [1,2], w którym zdefiniowano pięć języków, tzn. tekstowe ST, IL (*Structured Text*, *Instruction List*), graficzne LD, FBD (*Ladder Diagram, Function Block Diagram*) oraz mieszany język SFC (*Sequential Function Chart*). Do przykładowych środowisk programistyczno-wykonawczych implementujących te języki należą SIMATIC STEP 7 Siemensa [3], TwinCAT 3 Beckhoffa [4], czy Freelance Engineering z ABB [5]. Strukturę takiego środowiska można podzielić na trzy główne składniki, tj. środowisko programistyczne IDE (*Integrated Development Environment*), kompilator języków IEC 61131-3 oraz środowisko wykonawcze (*runtime*). W edytorach języków zawartych w IDE tworzony jest program sterowania, który następnie kompilator przekształca na wykonywalny kod binarny. Kod ten jest przenoszony do środowiska wykonawczego dedykowanego dla danego sterownika. Kod binarny jest wykonywany w czasie rzeczywistym z zadanym cyklem, a niekiedy w reakcji na określone zdarzenie.

Obecnie stosowane są trzy sposoby generacji kodu binarnego scharakteryzowane w [6,7]. W pierwszym kompilator bezpośrednio przetwarza program źródłowy z języka IEC 61131-3 na kod binarny procesora sterownika (tzw. kod natywny). Relatywnie prosty *runtime* wykonuje taki kod szybko, a sposób ten jest stosowany przez wielkoseryjnych producentów wymienionych wyżej. Jednak bezpośrednia kompilacja jest odpowiednia tylko dla jednego typu procesora. Zmiana CPU wymaga nowego kompilatora bądź przynajmniej nowej wersji dotychczasowego.

Drugi sposób obejmuje najpierw translację programu IEC 61131-3 do C/C++, a potem kompilator przekształca C/C++ do kodu binarnego platformy docelowej. Kompilatory C/C++ są ogólnodostępne więc podejście takie można implementować na różnych platformach. Odpowiada to szczególnie edukacji [8-10] korzystającej często z kompilatorów C/C++ typu *open-source*. Nie dotyczy to zwykle producentów korzystających z licencjonowanych kompilatorów (np. [11]), a ponadto ponoszących opłaty ze względu na komercyjne wykorzystanie.

Trzeci sposób bazuje na koncepcji maszyny wirtualnej, tzn. na pewnym nieistniejącym fizycznie procesorze, implementowanym wyłącznie programowo, którego zadaniem jest wykonywanie kodu pośredniego generowanego dla niego przez kompilator. Podejście takie zyskało na znaczeniu ze względu na powszechne wykorzystanie Javy i .NET z maszynami *Java Virtual Machine* (JVM) [12] i *Common Language Runtime* (CLR) [13]. Dla przykładu, w [14] program IEC 61131-3 jest kompilowany do kodu bajtowego Java (*Java bytecode*) przenoszonego następnie do urządzenia z wbudowaną maszyną JVM. Podobnie w [15] jeden moduł środowiska kompiluje program IEC 61131-3 do C#, a drugi implementuje go w CLR. Naturalnie podejście to umożliwia aplikacje wieloplatformowe, ale nie zwalnia producentów

sterowników od opłat za komercyjne wykorzystanie. Ponadto wymagana jest dostępność tych modułów dla platformy sprzętowej sterownika. Stanowi to przeszkodę zwłaszcza dla małych i średnich producentów.

Wymagania tego można uniknąć projektując całkiem nową, unikalną maszynę wirtualną z odpowiadającym jej językiem pośrednim. Nasuwającą się tutaj wprost możliwością jest wykorzystanie IL jako języka pośredniego [16], który jednak stwarza trudność w przypadku złożonych typów danych i programów [17,18]. Prawdopodobnie pierwszym rozwiązaniem niekorzystającym z IL było środowisko CPDev (*Control Program Developer*) rozwijane na Politechnice Rzeszowskiej od kilkunastu lat [19] stanowiące bazę dla niniejszej pracy. Jest ono przeznaczone dla niewielkich sterowników wykonujących programy IEC 61131-3 w ramach jednego zadania. Nieco później pojawiło się rozwiązanie opisane w [20], gdzie w architekturze maszyny wirtualnej występuje stos i rejestry podobne do x86, a język pośredni przypomina asembler. Maszyna została napisana w C z przeznaczeniem dla platformy ARM sterownika SoftPLC.

1.2. Cechy maszyn wirtualnych

Kod binarny wykonywany przez maszynę wirtualną reprezentuje określony program w języku pośrednim mającym pewien zasób instrukcji przeznaczonych do przetwarzania przyjętych typów danych. Wykonywanie kodu binarnego odbywa się z wykorzystaniem mechanizmów maszyny wynikających z jej architektury, używając do tego celu natywnych zasobów zastosowanej platformy sprzętowej, w szczególności CPU i pamięci. Do mechanizmów maszyny należą przede wszystkim moduł przetwarzania instrukcji oraz przeznaczona dla niej pamięć.

Rozwiązania bazujące na maszynie wirtualnej mają trzy istotne zalety. Po pierwsze kompilowany program źródłowy, tutaj IEC 61131-3, i program w języku pośrednim nie zależą od platformy sprzętowej. Po drugie implikuje to, że potrzebny jest tylko jeden kompilator, a nie kilka kompilatorów dla różnych CPU. Możliwe są zatem aplikacje wieloplatformowe. Trzecią zaletą jest wykonywanie programów w chronionym środowisku (izolowanym), co zapobiega propagacji błędów poza wyznaczone granice.

Jednakże zastosowanie maszyny wirtualnej ma również wady. Podstawową wadą jest wolniejsze wykonywanie kodu pośredniego niżby to było w przypadku kodu natywnego wygenerowanego przez kompilator dla platformy docelowej (pierwszy sposób powyżej). Wynika to głównie stąd, że instrukcje i dane kodu pośredniego muszą być dekodowane programowo, podczas gdy typowa jednostka CPU korzysta z dekoderów sprzętowych

i przetwarzania potokowego (*pipelining*) [21]. Nieunikniona jest także potrzeba dostosowania środowiska wykonawczego (*runtime*) do konkretnej platformy sprzętowej. Warto jednak zwrócić uwagę, że wolniejsze wykonywanie kodu pośredniego nie musi być istotnym ograniczeniem wobec wysokich częstotliwości zegarowych obecnych CPU i cyklu PLC wynoszącym nie mniej niż kilka milisekund.

1.3. Motywacja i cel

Jest zrozumiałe, że przystępując do opracowania środowiska programistyczno-wykonawczego zgodnego z normą IEC 61131-3 i zorientowanego na wieloplatformowe zastosowania poprzez wykorzystanie maszyny wirtualnej należało przyjąć dość istotne ograniczenia i założenia. Miało to naturalnie również miejsce w odniesieniu do wspomnianego środowiska CPDev [19], gdy wśród mikroprocesorów przeznaczonych dla małych sterowników dominował 8-bitowy AVR [22]. W związku z tym, w odniesieniu do maszyny wirtualnej przewidzianej dla CPDev postawiono dwa podstawowe założenia:

- 16-bitowa adresacja,
- bajtowy dostęp do danych.

16-bitowa adresacja ograniczyła rozmiar pamięci programu i danych do 64 kB, co wystarczało dla niewielkich aplikacji. Bajtowy dostęp do pamięci odpowiedni dla typu BOOL, podstawowego w zastosowaniach PLC, implikował potrzebę składania danych typu WORD, REAL, DATE_AND_TIME itd. z pojedynczo odczytanych bajtów. Według tych założeń opracowana została architektura maszyny wirtualnej oraz dotyczący jej język pośredni VMASM (*Virtual Machine Assembler*), a następnie kompilator języka źródłowego ST do VMASM [19,23]. Maszynę tę napisano w C i tutaj jest ona oznaczana jako VM16.

Ze względu na możliwe zastosowania praktyczne przyjęto także, że implementując maszynę VM16 na konkretnej platformie sprzętowej korzystać się będzie wyłącznie z oficjalnych narzędzi programistycznych udostępnianych wraz z procesorem zastosowanym na tej platformie, a nie z dodatkowych narzędzi zewnętrznych, z których użyciem związany jest koszt. Przyjęto, że za całość oprogramowania będą odpowiadać autorzy, co często określane jest jako *bare-metal*.

Należy jeszcze dodać, że 16-bitowa adresacja dotyczy tylko maszyny wirtualnej, a nie natywnej adresacji w platformie docelowej, bo tę określa kompilator C. Dlatego oryginalnie maszynę VM16 można było także implementować na procesorach 16- i 32-bitowych [6].

Środowisko CPDev z maszyną VM16 zastosowano najpierw w mini-systemie DCS ogólnego przeznaczenia produkowanym wtedy przez LUMEL Zielona Góra [24]. Drugim

zastosowaniem były i są podsystemy automatyzacji statków produkowane przez holenderską firmę Praxis Automation [25] realizujące zarządzanie energią, ustawianie napędów, ochronę przeciwpożarową itp. Trzecim zastosowaniem CPDev są moduły telesterujące dla stacji energetycznych i rekonfiguracji sieci z hiszpańskiej firmy iGrid [26]. Istotną rolę spełniają w nich programy aktywowane zdarzeniami (*trigger-mode*).

W międzyczasie w nowych konstrukcjach sterowników zaczęto stosować procesory o architekturze ARM. Dostosowanie kompilatora CPDev i maszyny wirtualnej do adresacji 32-bitowej było naturalne, bo od początku przewidywano w nich parametr *AddressSize*. Powstała więc nowa maszyna VM32 z rozmiarami pamięci programu i danych sięgającymi 4 GB. Badania porównawcze przeprowadzone przez Praxis pokazały jednak, że VM32 jest wyraźnie mniej wydajna (w sensie czasu wykonywania programu) niż dotychczasowa VM16 przy implementacji na tej samej platformie. Okazało się, że powodem jest bajtowy dostęp do pamięci wymagający składania złożonych danych. Dla rozwiązania problemu, zamiast dostępu bajtowego zastosowano wówczas kopiowanie za pomocą standardowej funkcji *memcpy* z języka C/C++. Powoduje to przeniesienie wyboru optymalnej metody kopiowania na kompilator i bibliotekę standardową. Po zastosowaniu *memcpy* w maszynach VM16 i VM32 wydajności ich stały się niemal identyczne.

Trzecią zastosowaną metodą dostępu do pamięci jest dostęp bezpośredni za pomocą wskaźników. Okazał się on szczególnie odpowiedni dla adresacji z wyrównaniem 4-bajtowym (*4-byte alignment*) stosowanej w niektórych architekturach ARM [27]. Adresacja taka wymaga jednak większej pamięci, bo np. na zmienną typu BOOL należy przeznaczyć cztery bajty a nie jeden. W metodzie tej potrzebne były również nowe instrukcje VMASM dla obsługi 4-bajtowego dostępu, a więc modyfikacje kompilatora i maszyny wirtualnej. Maszyna 32-bitowa dla danych z 4-bajtowym wyrównaniem jest oznaczona tutaj jako VM32A. Z zastosowania wyrównania w maszynie VM16 zrezygnowano ze względu na wzrost rozmiaru pamięci.

W sumie, w środowisku CPDev zależnie od architektury i skali problemu sterowania można zastosować maszyny wirtualne VM16, VM32 i VM32A, przy czym w dwóch pierwszych możliwy jest dostęp bajtowy, kopiowanie *memcpy* oraz dostęp bezpośredni. Natomiast VM32A wymaga wyrównania danych. Bliższe zbadanie wydajności tych wariantów wraz z rozmiarami pamięci jest jedną z motywacji niniejszej pracy.

Środowisko CPDev od początku było przeznaczone do wykonywania jednego zadania ze stałym cyklem, co wystarczało dla podstawowych zastosowań. W przypadku sterowania logiczno-sekwencyjnego typu PLC standardowym cyklem jest 10 ms, a w przypadku regulacji PID – 50 lub 100 ms. Gdyby sterowany obiekt wymagał zarówno sterowania PLC i jak

i regulacji PID, to potrzebne byłyby dwa sterowniki, bądź dostęp do stacji procesowej DCS z wielozadaniowym systemem operacyjnym czasu rzeczywistego RTOS (*Real Time Operating System*). Pewnym rozwiązaniem mógłby być także sterownik jednozadaniowy, ale uzupełniony o szeregowanie wykonywanych programów, np. w formie aktywacji PID co 5 lub 10 cykli PLC.

Tymczasem przynajmniej od dekady dostępne są już procesory wielordzeniowe, jak np. wykorzystany w popularnym mikrokomputerze Raspberry Pi [28] i inne, które niedawno pojawiły się w nowych sterownikach [29-31]. Każdy z rdzeni takiego procesora może wykonywać własny program, a do wymiany danych między nimi służy pamięć współdzielona. Sterownik dwurdzeniowy mógłby więc funkcjonować jako PLC+PID, PLC+I/O, PLC+HMI (*Human Machine Interface*) itp., a trzyrdzeniowy np. jako PLC+PID+HMI. Na tym tle zrodziła się potrzeba przeanalizowania, czy środowiska CPDev nie można byłoby rozszerzyć na dwa lub więcej rdzeni.

Po wstępnej analizie okazało się, że obecny kompilator ST→VMASM i środowisko IDE będą wymagały tylko niewielkich uzupełnień pod warunkiem zachowania odpowiedniego trybu programowania rdzeni, natomiast znaczących uzupełnień należy oczekiwać po stronie maszyny wirtualnej. Stanowi to więc podstawową motywację i dlatego:

Głównym celem niniejszej pracy jest opracowanie metody rozszerzenia maszyny wirtualnej przeznaczonej dla środowiska jednordzeniowego na środowisko wielordzeniowe, w którym rdzenie wykonują programy IEC 61131-3 mogące wymieniać między sobą dane.

Zakłada się przy tym, że funkcjonowanie maszyny wirtualnej oparte będzie wyłącznie na rozwiązaniach programistycznych dostarczonych wraz z procesorem wielordzeniowym, czyli wspomniany wyżej tryb *bare-metal*. Wyklucza to użycie systemu operacyjnego w odniesieniu do rdzeni realizujących sterowanie. System taki, np. Linux czy FreeRTOS, zaimplementowany na odrębnym rdzeniu może być użyty do zadań dodatkowych, nie wymagających ściśle terminowego wykonywania, jak komunikacja, diagnostyka, obliczenia wspierające itp. W przypadku środowiska CPDev w rdzeniach przeznaczonych do sterowania może funkcjonować każda z maszyn VM16, VM32 i VM32A z wybranym trybem dostępu do pamięci. Stąd kolejną motywacją rzutującą na zawartość pracy byłoby potwierdzenie osiągnięcia celu poprzez praktyczną weryfikację projektów IEC 61131-3 implementowanych zarówno w dwurdzeniowym mikrokontrolerze, jak i w bardziej złożonym systemie z komunikacją, zawierającym np. czterordzeniowy mikrokomputer Raspberry Pi.

1.4. Przegląd treści

W następnym rozdziale przedstawiono ogólną charakterystykę środowiska CPDev, w tym zwłaszcza jego część dotyczącą kompilacji. Składają się na nią translatory graficznych języków LD, FBD i mieszanego SFC do tekstowego języka ST, oraz kompilator języków ST i IL do pośredniego języka VMASM, z którego generator kodu tworzy wykonywalny kod binarny. Język VMASM jest zorientowany na normę IEC 61131-3 co oznacza, że pierwsza część jego instrukcji odpowiada wprost standardowym funkcjom normy [6,7,19]. Drugą część stanowią tzw. procedury systemowe wynikające z architektury maszyny wirtualnej. Do przetwarzania instrukcji VMASM na kod binarny służą identyfikatory cyfrowe, które w przypadku funkcji składają się z identyfikatora ich grupy, liczby wejść (operandów) oraz identyfikatora typu danych. Podano przykłady definicji kilku funkcji i procedur systemowych języka VMASM. Na informacje uzupełniające składa się wykaz plików pakietu CPDev oraz jego dwie aplikacje dla PC.

Część wykonawcza środowiska CPDev, czyli maszyna wirtualna, jest omówiona w rozdziale 3. Na architekturę maszyny składa się moduł przetwarzania instrukcji VMASM, pamięci kodu i danych, stosy, rejestry oraz interfejs platformy docelowej [6,23]. Na podstawie identyfikatorów grupy i typu moduł przetwarzania wybiera instrukcję, którą następnie wykonuje. Wybór instrukcji przedstawiono zarówno w formie algorytmu graficznego jak i semantycznego modelu denotacyjnego [32,33], wyrażającego znaczenie algorytmu. Łatwiej jest na podstawie takiego modelu napisać program w praktycznie dowolnym języku, który w zoptymalizowanej formie nie od razu bywa zrozumiały. Następne przykłady modeli denotacyjnych i ich implementacji w C/C++ znajdują się w Dodatku A. Do uruchamiania skompilowanego programu służy oryginalny symulator CPSim lub nowe środowisko IDE zintegrowane z edytorami języków, symulacją *online* i rozszerzonym debugowaniem. Interfejs platformy docelowej wiąże maszynę wirtualną z rozwiązaniami sprzętowymi. Oryginalnie ma on postać zbioru prototypów funkcji niskopoziomowych [23], które inżynier wdrażający maszynę wirtualną wypełnia odpowiednią treścią. Znajdują się wśród nich prototypy funkcji `VMP_PreCycle` i `VMP_PostCycle` wykonywanych na początku i końcu cyklu sterowania, które odpowiadają za obsługę wejść i wyjść.

W rozdziale 4 przeanalizowano wpływ różnych metod dostępu do pamięci na wydajność maszyn wirtualnych VM16, VM32 i VM32A, rozumiejąc przez to czas wykonywania programów testowych [34]. Poziomem odniesienia jest dostęp bajtowy BYTE_ACCESS z maszyny oryginalnej. Jak sygnalizowano wyżej, 32-bitowa maszyna VM32 ustępuje wtedy wyraźnie 16-bitowej VM16, wobec niewykorzystywania przyspieszających

mechanizmów sprzętowych. Dla metody MEMCPY_ACCESS stosującej standardową w C/C++ funkcję *memcpy*, maszyny VM16 i VM32 okazują się równie wydajne, przy czym wydajność VM16 wyraźnie wzrosła w stosunku do BYTE_ACCESS. Bezpośrednia (wskaźnikowa) metoda DIRECT_ACCESS w trybie z 4-bajtowym wyrównaniem ALIGN_4B okazała się jeszcze trochę wydajniejsza. Wymagało to jednak dodania kilku nowych instrukcji VMASM zapewniających wyrównanie, a więc potrzebę modyfikacji kompilatora, bibliotek i samej 32-bitowej maszyny wirtualnej. Odwrotnie niż wydajność wyglądają natomiast rozmiary pamięci programów testowych dla różnych maszyn. Kod jednego z nich dla maszyny VM32 okazuje się o 50% większy niż dla VM16, a kod dla VM32A nawet dwukrotnie większy. W tej sytuacji wybór jednej z trzech maszyn z określoną metodą dostępu do pamięci powinien zależeć od przeznaczenia projektowanego sterownika.

Koncepcja rozszerzenia środowiska programistyczno-wykonawczego przeznaczonego oryginalnie dla jednozadaniowego projektu typu PLC lub PID na środowisko obsługujące dwa rdzenie, wykonujące dwa projekty jako zadania, jest przedstawiona w rozdziale 5. Środowiskiem takim może być, ale nie musi, środowisko CPDev. Zamiast systemu operacyjnego RTOS, który mógłby koordynować zadania, tutaj wymianę danych między rdzeniami ma zapewnić pamięć współdzielona. Według normy IEC 61131-3 do wymiany danych między zadaniami służą zmienne globalne. W tej sytuacji oraz po analizie dwóch środowisk z systemami RTOS [4,5] na koncepcję rozszerzenia jednozadaniowego środowiska na dwa zadania-projekty implementowane w osobnych rdzeniach składają się następujące zasady [35]:

- jednakowe listy deklaracji w obydwu projektach zmiennych globalnych wymienianych między rdzeniami,
- uzupełnienie deklaracji każdej z wymienianych zmiennych o atrybut określający formę dostępu do pamięci współdzielonej, np. jako READ i WRITE,
- wymiana zmiennych poprzez pamięć współdzieloną na początku i końcu cyklu danego rdzenia,
- dostęp do pamięci współdzielonej przez dany rdzeń w sposób wykluczający kolizję z drugim rdzeniem.

Dwie pierwsze zasady dotyczą części programistycznej środowiska, a trzecia i czwarta – części wykonawczej, czyli tutaj maszyny wirtualnej. Zasady te odnoszą się również do procesora z większą liczbą rdzeni pod warunkiem, że atrybut WRITE nadawany jest zmiennej tylko w jednym rdzeniu.

Rozszerzenie środowiska CPDev według powyższej koncepcji wraz z praktyczną implementacją w dwurdzeniowym mikrokontrolerze STM32 [36] przedstawiono w rozdziale 6. Okazuje się, że jedynym zewnętrznym rozszerzeniem IDE są atrybuty READ/WRITE identyfikujące wymieniane zmienne globalne, bo jednakowe listy ich deklaracji w obydwu projektach wymagają tylko uwagi programisty. W zamian można też utworzyć jeden wspólny projekt przeznaczony dla dwóch rdzeni, a potem wydzielić z niego dwa podprojekty implementowane w rdzeniach. W sumie „dwurdzeniowy" CPDev wygląda z zewnątrz niemal tak samo jak oryginalny. Rozszerzenia wymaga natomiast język pośredni VMASM o procedury zapisu i odczytu do/z pamięci współdzielonej, co pociąga za sobą uzupełnienia w kompilatorze, a szczególnie w maszynie wirtualnej. Procedury te muszą zawierać zabezpieczenia przeciw kolizji przy dostępie do tej pamięci. Do symulacji dwóch projektów wykonywanych z różnymi cyklami, jednego napisanego w ST, drugiego w FBD, wykorzystano wielowątkowy WinController [37]. Do implementacji praktycznej posłużyła płyta uruchomieniowa serii Nucleo z mikrokontrolerem STM32H755 [38]. Bezkolizyjny dostęp do pamięci współdzielonej zapewniają dodatkowe funkcje wykonywane w fazach *precycle* i *postcyle* cyklu maszyny, wykorzystujące sprzętowy semafor mikrokontrolera.

Bardziej rozbudowany projekt jest przedstawiony w rozdziale 7, gdzie chodzi o identyfikację znaczników RFID (transponderów) [39] rozmieszczonych na płycie jako siatka kwadratowa. Do ich odszukania służy mały dydaktyczno-eksperymentalny robot mobilny [40] z dodatkowo zainstalowanym czytnikiem RFID, enkoderami i zmodyfikowanym oprogramowaniem, sterowany przez dwurdzeniowy mikrokontroler ESP32 [41]. Przemieszczaniem robota wzdłuż siatki i całością systemu zarządza czterordzeniowy Raspberry Pi [28] komunikujący się z ESP32 poprzez Wi-Fi. Zbliżona wersja systemu, ale ze specjalnie skonstruowanymi mikrorobotami klasy (2,0) oraz oprogramowaniem napisanym w C/C++ została omówiona w [42]. W przedstawionym tutaj rozwiązaniu dwa rdzenie Raspberry Pi z maszynami wirtualnymi CPDev1, CDev2 kontrolują eksplorację obszaru, rejestrują identyfikatory znaczników RFID i poprzez Wi-Fi za pośrednictwem trzeciego rdzenia wysyłają komendy dotyczące ruchu robota do maszyny CPDev3 w jednym z dwóch rdzeni ESP32. Wykonywanie tych komend, tzn. sterowanie silnikami, prowadzi niskopoziomowe oprogramowanie robota umieszczone w drugim rdzeniu ESP32. W oprogramowaniu tym interpretację pomiarów prowadzą funkcje *firmware*, natomiast sterowanie ruchem odpowiednio do komend zostało opracowane od podstaw. Oprogramowanie ST, sterujące w stanie normalnym całym systemem, utworzono jako jeden wspólny projekt, przedstawiając funkcjonalność maszyn CPDev1, CPDev2, CPDev3 oraz wyniki symulacyjne. Kody ST jednostek oprogramowania POU znajdują się w Dodatku B. Po podziale wspólnego projektu

na trzy podprojekty, maszyny CPDev1, CPDev2 w Raspberry Pi komunikują się poprzez pamięć współdzieloną, a CPDev3 z CPDev1, CPDev2 poprzez Wi-Fi. Charakteryzując architekturę oprogramowania wskazano na elementy wspólne, sposób implementacji maszyn CPDev1, CPDev2 za pomocą systemu Linux, maszyny CPDev3 za pomocą FreeRTOS oraz na składniki niskopoziomowego oprogramowania robota. Rozdział kończy opis postępowania w sytuacji nienormalnej, tzn. gdy wobec braku systemu lokalizacji typu GNSS RTK, robot na skutek różnego typu niedokładności i zakłóceń zboczył z wyznaczonej ścieżki, a nie napotkawszy spodziewanego znacznika RFID zatrzymał się sygnalizując pominięcie. Potrzebna jest wtedy interwencja zewnętrzna w formie ustawienia robota nad ostatnio odczytanym znacznikiem i wznowienie procesu identyfikacji. Za obsługę takiej sytuacji odpowiada oprogramowanie niskopoziomowe.

W ostatnim rozdziale podsumowano wyniki rozprawy. Badania programów testowych pokazały, że wydajność przedstawionych wersji maszyny wirtualnej silnie zależy od metody dostępu do pamięci, mającej także wpływ na rozmiar pamięci programu. W miarę proste zasady składające się na ogólną koncepcję środowiska wielordzeniowego, poczynając od wspólnej listy wymienianych zmiennych globalnych, uwzględniają zróżnicowanie cyklu programów wykonywanych przez maszyny wirtualne w rdzeniach. Z punktu widzenia programisty, środowisko CPDev rozszerzone na aplikacje wielordzeniowe tylko nieznacznie różni się od środowiska oryginalnego. Dwa praktyczne przykłady pokazały, że sterownik wielordzeniowy byłby w stanie zastąpić konwencjonalny sterownik z wielozadaniowym systemem operacyjnym RTOS. Wniosek ten określa zarazem zasadniczy użytkowy rezultat rozprawy.

2. Środowisko programistyczno-wykonawcze CPDev

Poniższa charakterystyka środowiska CPDev rozpoczyna się od kroków przetwarzania programu sterowania napisanego w jednym z języków normy IEC 61131-3 do kodu binarnego dla maszyny wirtualnej będącej środowiskiem wykonawczym [6,7]. Językiem pośrednim maszyny jest VMASM, którego instrukcje dzielą się na funkcje odpowiadające wprost normie IEC 61131-3 oraz procedury asemblera wynikające z architektury maszyny. Skaner i parser kompilatora CPDev przetwarzają program źródłowy na kod mnemoniczny w języku VMASM. Następnie generator kodu przekształca ten kod na kod binarny zastępując mnemoniki instrukcji właściwymi identyfikatorami cyfrowymi oraz przydzielając adresy zmiennym. W przypadku przeciążonej funkcji rozszerzalnej, jak np. ADD, identyfikator ten zawiera również informację o typie danych i liczbie operandów. Wymienione kroki ilustrowane są prostym programem załączania i wyłączania silnika, z zabezpieczeniem przed ponownym zbyt szybkim załączeniem. Rozdział kończy się wykazem plików pakietu CPDev oraz krótkimi charakterystykami jego laboratoryjnej wersji CPCtrl oraz wielowątkowego WinControllera.

2.1. Charakterystyka ogólna

Sterowniki programowalne, znane również jako PLC lub PAC (*Programmable Automation Controller*), są najczęściej stosowanymi urządzeniami sterującymi procesami produkcyjnymi. Monitorując stan wejść (czujniki, przetworniki), na podstawie cyklicznie wykonywanych programów sterują stanem wyjść (zawory, siłowniki itd.). Choć różnice między sterownikami PLC i PAC coraz bardziej się zacierają, to sterowniki PAC można uważać za rozbudowane wersje sterowników PLC. Wśród urządzeń sterujących wykorzystywanych w przemyśle popularne stają się również komputery przemysłowe IPC. Są one wydajniejsze i elastyczniejsze niż PAC, pracują w oparciu o rozbudowane systemy operacyjne, takie jak Linux lub Windows, a ich implementacja jest często bardziej pracochłonna.

Obowiązująca norma IEC 61131 [1,2] (w Polsce jako PN-EN 61131) ustandaryzowała aplikacje systemów sterowania w podstawowych aspektach. Są nimi wymagania dotyczące części sprzętowej sterownika, unifikacja języków programowania i wymiany informacji, oraz sposób projektowania systemów sterowania. Szczególne znaczenie ma część trzecia normy (IEC 61131-3) definiująca pięć języków programowania, tzn. tekstowe ST, IL, graficzne LD, FBD oraz język mieszany SFC. Oprócz architektury systemu sterowania, modelu komunikacyjnego oraz elementów konfiguracji, część ta definiuje również typy danych oraz przedstawia syntaktykę, semantykę i pragmatykę kodowania.

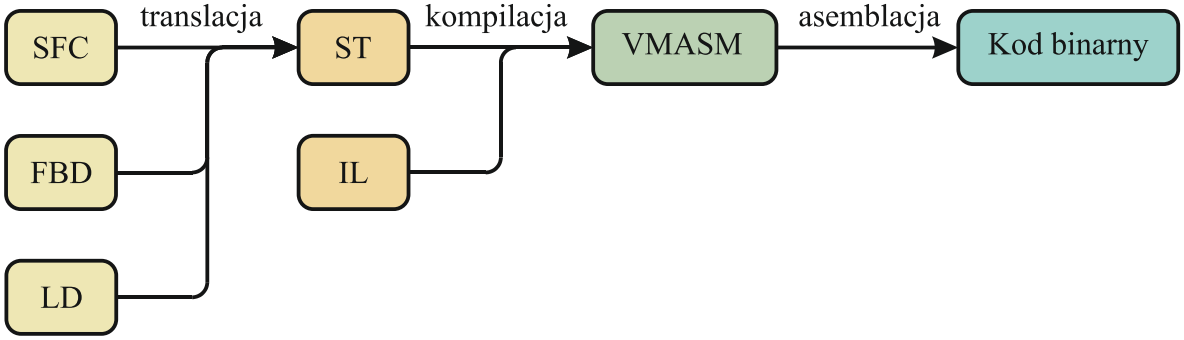
Jak podano we Wprowadzeniu, środowiska inżynierskie zgodne z normą IEC 61131-3 składają się z trzech zasadniczych komponentów, tj. środowiska programistycznego IDE, kompilatora języka źródłowego oraz środowiska wykonawczego *runtime*. Kompilator tłumaczy program źródłowy napisany w jednym z języków normy na kod binarny przesyłany do środowiska wykonawczego procesora sterownika. Środowisko to wykonuje kod w czasie rzeczywistym z zadanym cyklem. Przykładami pakietów oprogramowania o takiej strukturze są wspomniane wcześniej STEP 7, TwinCAT 3 i Freelance Engineering [3-5].

Rozważane w niniejszej pracy środowisko CPDev jest zgodnym z normą IEC 61131-3 rozwiązaniem obsługującym jej pięć języków, a ponadto umożliwia projektowanie paneli operatorskich HMI (*Human-Machine Interface*) [43], generowanie dokumentacji oraz zawiera rozszerzenia wspierające diagnostykę i badania. Podstawowe funkcjonalności są zbliżone do wymienionych wyżej pakietów, natomiast wyróżnia go otwartość na integrację z różnymi platformami sprzętowymi oraz systemami komunikacyjnymi. Możliwe jest uruchamianie programów na platformach z niewielkimi mikrokontrolerami, jak np. STM32 [36] czy ESP32 [41], aż do zaawansowanych procesorów typu x86 i x64.

Rozproszone systemy sterowania o niejednorodnej architekturze sprzętowej, tzn. ze sterownikami zawierającymi odmienne procesory, wymagają zwykle odrębnych narzędzi programistycznych i środowisk wykonawczych. Pojawia się wówczas problem przenośności oprogramowania między różnymi platformami. Jego rozwiązaniem może być system programowania oparty o maszynę wirtualną [12,13]. W rozwiązaniu tego typu kod źródłowy jest kompilowany do uniwersalnego kodu pośredniego (*intermediate code*), niezależnego od procesora lub platformy sprzętowej. Maszyna wirtualna jest elementem środowiska wykonawczego, które umożliwia wykonywanie tego kodu za pomocą mechanizmów danej platformy. Jak już podawano (pkt. 1.1), w [14] programy ST i FBD kompilowane są do kodu bajtowego Java przesyłanego do urządzeń z maszyną JVM. Podobnie w [15], jeden moduł kompiluje projekty IEC 61131-3 do C#, a inny wdraża je na maszynach wirtualnych CLR w systemach Windows lub Linux. W tych przypadkach generowany kod bajtowy jest wspólny dla różnych procesorów, dzięki czemu staje się on przenośny między różnymi platformami. Wadą takiego rozwiązania jest natomiast wolniejsze wykonywanie kodu w stosunku do równoznacznego kodu natywnego. Oprócz sterownika SoftPLC [20] wspomnianego we Wprowadzeniu, do przemysłowych rozwiązań implementujących maszyny wirtualne należy ISaGRAF [44] z *Target Independent Code* oraz środowisko STRATON [45], którego szczegóły nie są jednak ujawniane. Jak podawano, IL jest językiem pośrednim w rozwiązaniach akademickich [16-18] i edukacyjnych [46,47]. Maszyna wirtualna zastosowana w środowisku

CPDev została zaprojektowana w sposób spójny z ideą JVM, zorientowano ją jednak na niewielkie sterowniki i systemy wbudowane.

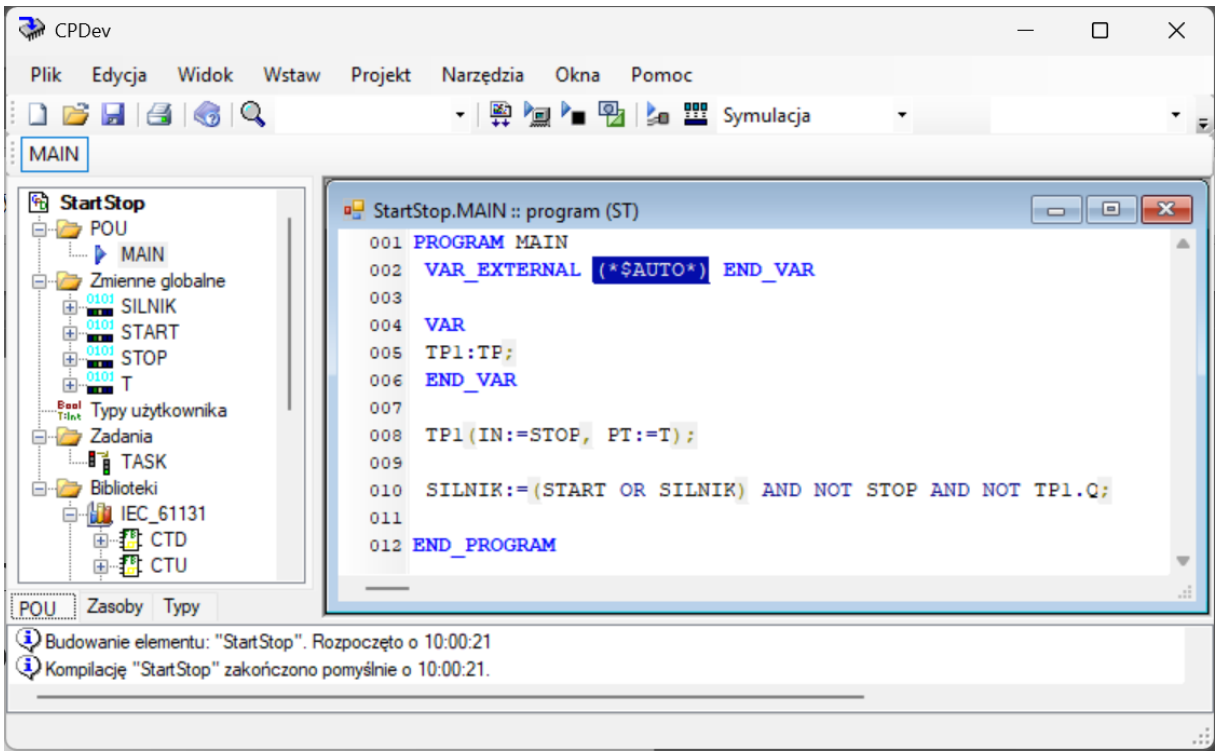
Etapy przetwarzania programu sterowania przez kompilator CPDev utworzony w języku C# pokazano na rysunku 2.1 [6,7]. Dla diagramów w językach FBD, LD i SFC następuje najpierw translacja do języka ST. Następnie programy ST lub IL kompilowane są do mnemonikowego języka pośredniego VMASM opisanego dalej. Ostatecznie asemblacja konwertuje je na kod binarny przesyłany do maszyny wirtualnej. Maszyna (*runtime*) opracowana w języku C/C++ może być implementowana na różnych platformach sprzętowych.



Rys. 2.1. Przetwarzanie programów sterowania w środowisku CPDev.

Jak podano we Wprowadzeniu, maszyna wirtualna CPDev wykonująca instrukcje kodu binarnego występuje obecnie w trzech wersjach, tzn. 16-, 32-bitowej (VM16, VM32) oraz 32-bitowej z adresowaniem wyrównującym długość danych (VM32A) [48]. Zróżnicowanie rozmiaru i sposobu adresowania umożliwia implementacje wersji maszyny w różnorodnych układach (rozdz. 4). Dla małych sterowników przewidziana jest maszyna VM16 ograniczona adresowaniem do 64 kB pamięci programu i danych. Systemy wymagające rozbudowanych programów oparte są o wersję VM32, z adresacją do 4 GB pamięci programu i danych. Rozmieszczenie wyrównujące w VM32A powoduje opuszczanie bajtów pamięci dla zmiennych o rozmiarze mniejszym od długości 4-bajtowego słowa (pkt. 4.1). W szczególnym przypadku BOOL pomijane są 3 bajty.

Na rysunku 2.2 przedstawiono prosty program sterujący napisany w języku ST w środowisku CPDev IDE. Celem programu jest sterowanie silnikiem z zabezpieczeniem przed przedwczesnym ponownym załączeniem, które przy wielokrotnym powtarzaniu groziłoby awarią. Lokalny czasomierz TP1 (*Pulse Timer*) odlicza czas od momentu aktywacji wejścia STOP i dopiero po upływie czasu T silnik udaje się uruchomić ponownie. Okno symulatora CPSim oraz okna symulacji IDE dotyczące tego przykładu pokazano w następnym rozdziale.



Rys. 2.2. Okno edycji programu ST w środowisku CPDev IDE.

2.2. Język pośredni VMASM

Program w języku pośrednim VMASM otrzymany w wyniku kompilacji źródłowego programu ST lub IL służy do generowania kodu binarnego uruchamianego w sterowniku. Interpreterem tego kodu jest maszyna wirtualna. Język VMASM został opracowany zgodnie z wytycznymi normy IEC 61131-3. Podstawą działania są zmienne lub stałe przechowujące wartości pod konkretnymi adresami w pamięci. Elementarne typy danych wraz z ich rozmiarem podano w tabeli 2.1. Zaimplementowano również wielowymiarowe tablice oraz struktury proste i zagnieżdżone. Liczba typów obsługiwanych przez konkretną aplikację może być ograniczona poprzez parametryzację kompilatora.

Tab. 2.1. Typy danych i rozmiar pamięci w bajtach.

Typ	Rozmiar
`BOOL, BYTE, SINT, USINT`	1
`INT, UINT, WORD`	2
`REAL, DINT, UDINT, DWORD`	4
`DATE, TIME, TIME_OF_DAY`	4
`LREAL, DT, LINT, ULINT, LWORD`	8
`STRING, WSTRING`	zmienny

Składnia i semantyka pojedynczej instrukcji języka VMASM wygląda następująco [7]:

```
[:etykieta] instrukcja [operand1][,operand2]...
```
(2.1)

Etykieta początkowa jest elementem opcjonalnym i służy jako argument, np. w instrukcjach skoków. Konkretna instrukcja może wymagać operandów, zwykle zmiennych lub stałych. W przypadku funkcji generujących wartości, `[operand1]` oznacza zmienną wynikową, czyli faktyczny adres, więc powyższa składnia wyraża operacje typu „pamięć do pamięci" (*memory-to-memory*). Ze względu na zróżnicowanie rozmiarów typów danych maszyna wirtualna CPDev nie została wyposażona w akumulator. Symbol specjalny ? (pytajnik) wykorzystywany jest do tworzenia identyfikatorów (nazw) nieobjętych normą IEC 61131-3, a które są potrzebne przy kompilacji. Zmienne tymczasowe VMASM tworzone przez kompilator poprzedza przedrostek ?LR?. Operator kropki (.) pozwala na dostęp do elementów struktur. Wartości bezpośrednie zapisywane są w postaci szesnastkowej zaczynając od znaku # (zwykle w formie *Little Endian*).

Można dodać, że w języku IL wynik polecenia jest przechowywany w rejestrze tymczasowego wyniku CR (*Current Result*) równoważnym akumulatorowi [1,2]. Następnie CR jest kopiowany do zmiennej. Umieszczając wynik w pierwszym operandzie język VMASM łączy więc te dwa kroki w jeden. Jak wspomniano powyżej, pojęcie akumulatora nie istnieje w VMASM.

Instrukcje VMASM składają się z funkcji i procedur systemowych, których przykłady podano w tabeli 2.2. Funkcje są zgodne z normą IEC 61131-3 dopuszczając do 16 operandów, gdzie pierwszy oznacza wynik (`[operand1]`). W przeciwieństwie do funkcji, procedury systemowe nie zwracają wartości lub zwracają więcej niż jedną. Procedury kontrolują przebieg programu, obsługują pamięć, wywołują podprogramy itp. Z punktu widzenia programisty nie ma istotnej różnicy w używaniu funkcji czy procedur. Operacje arytmetyczne wykonywane są w ograniczonym zakresie liczbowym zależnie od typu. W przypadku liczb całkowitych jest to (-128, 127) dla SINT, (-32768, 32767) dla INT itd.

Tab. 2.2. Wybrane funkcje i procedury systemowe języka VMASM.

Funkcja		**Procedura systemowa**	
`EXPT`	Potęgowanie	`JMP`	Skok bezwarunkowy
`NEG`	Negacja	`JZ` `JNZ`	Skok warunkowy
`MUL`	Mnożenie	`JR`	Skok bezwarunkowy względny
`DIV`	Dzielenie	`JRN`	Warunkowy skok względny
`ADD`	Dodawanie	`CALB`	Wywołanie podprogramu

SUB	Odejmowanie	RETURN	Powrót z podprogramu
GT GE	Większy, Większy lub równy	GARD	Kopiowanie pamięci globalnej do obszaru lokalnego
LT LE	Mniejszy, Mniejszy lub równy	GAWR	Kopiowanie pamięci lokalnej do obszaru globalnego
EQ	Równy	FPAT	Wypełnianie bloku pamięci
NE	Różny	MEMCP	Kopiowanie bloku pamięci
AND	Iloczyn logiczny	MCD	Inicjowanie danych
CONCAT	Łącznie łańcuchów znakowych	TRML	Zakończenie cyklu obliczeniowego

2.3. Kompilator CPDev

Przenaszalność implementacji maszyny wirtualnej zależy przede wszystkim od rozmiaru adresu obsługiwanego przez procesor, czyli 16 lub 32 bity. Ponadto, niektóre typy danych mogą być zbędne w danym zastosowaniu. Aby zapewnić elastyczność rozwiązania, kompilator CPDev można parametryzować za pomocą pliku konfiguracyjnego LCF (*Library Configuration File*), którego przykładowy fragment pokazano w tabeli 2.3 [6].

Tab. 2.3. Przykładowa parametryzacja kompilatora w pliku LCF.

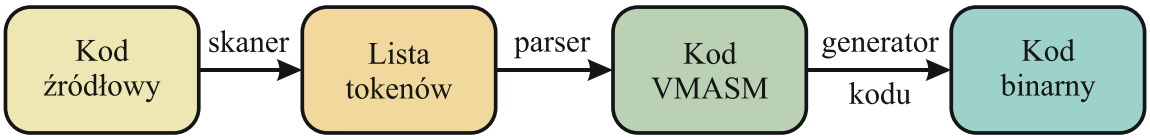
```
<HARDWARE>
  <AddressSize>
    2
  </AddressSize>
  <MaxCodeAddress>
    0xFFFF
  </MaxCodeAddress>
  <MaxDataAddress>
    0xFFFF
  </MaxDataAddress>
</HARDWARE>
```

```
<TYPES>
  <type name="UINT"
        implement="alias">
    <alias name="WORD"/>
  </type>
  <deny-type name="LREAL"/>
</TYPES>
```

Część `HARDWARE` zawiera rozmiar adresu `AddressSize` oraz maksymalne adresy `MaxAddress` kodu i danych. `AddressSize` o wartości 2 lub 4 bajty oznacza odpowiednio adresowanie 16-bitowe (VM16, 64 kB pamięci) i 32-bitowe (VM32, 4 GB). Należy zaznaczyć, że `AddressSize` definiuje adresowanie dotyczące wyłącznie maszyny wirtualnej i nie jest ściśle powiązany z platformą sprzętową. Z tego powodu maszyna VM16 może zostać zaimplementowana na procesorach 16-, 32- lub 64-bitowych.

Część `TYPES` pliku LCF definiuje niektóre typy danych jako alias oraz typy wyłączone z kompilacji poprzez `deny-type`, np. LREAL, o ile takie typy nie będą używane w prostszych programach.

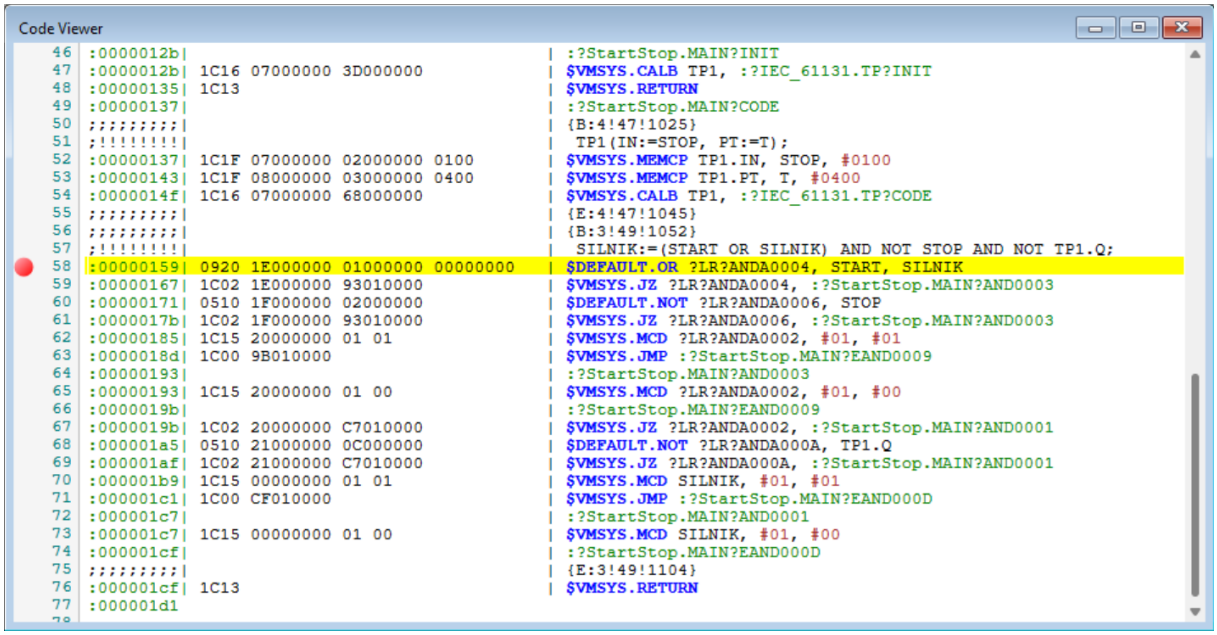
Zadaniem kompilatora CPDev jest konwersja pliku źródłowego zawierającego kod w języku ST (lub IL) na wykonywalny kod binarny. Uproszczony schemat działania kompilatora pokazano na rysunku 2.3 [7]. W pierwszym kroku skaner analizuje strumień znaków z pliku źródłowego i rozkłada go na tokeny leksykalne pogrupowane w kategorie, takie jak identyfikatory, słowa kluczowe, operatory i stałe. Tokeny z kategoriami gromadzone są na liście i przekazywane do parsera. Parser najpierw sprawdza, czy tokeny zawierają prawidłowe znaki, a następnie grupuje je w konstrukcje zgodnie z gramatyką języka ST. Jeśli konstrukcje są poprawne, analizator składni buduje wewnętrzne abstrakcyjne drzewo składni [49]. Drzewo to poddawane jest walidacji pod względem zmiennych, wyrażeń, wywołań i zagnieżdżeń poprzez wykonanie operacji kontrolnych [9,50]. Mając zweryfikowane drzewo, parser zastępuje konstrukcje ST zestawami mnemonicznych instrukcji VMASM wykorzystując elementarne typy danych i listę instrukcji. Kod VMASM jest konsolidowany z innymi kodami mnemonicznymi (np. z dodatkowych bibliotek) i zapisywany w specjalnym formacie tekstowym.



Rys. 2.3. Schemat działania kompilatora kodu ST na VMASM.

Kod VMASM odpowiadający programowi ST realizującemu sterowanie silnikiem (rys. 2.2) pokazano z prawej strony rysunku 2.4. Występują w nim procedury systemowe CALB, RETURN, MEMCP, JZ, MCD i JMP oraz funkcje OR i NOT. Wywołanie czasomierza TP1 (linia 8 w ST na rys. 2.2) zostało skompilowane do linii 52-54 kodu VMASM (linię 8 z ST powtórzono w linii 51 jako komentarz). Za pomocą procedury MEMCP kopiowane są wartości zmiennych STOP i T do wejść IN i PT czasomierza (ostatnim argumentem jest rozmiar w bajtach jako *Little Endian*, czyli 1 i 4), po czym CALB wywołuje podprogram w bibliotece IEC_61131 z jego obsługą. Zestaw mnemonicznych odpowiedników linii 10 w ST (powtórzonej w 57) rozpoczyna się od linii 58 w VMASM, gdzie w zmiennej tymczasowej `?LR?AND0004` (przedrostek ?LR?) określana jest wartość alternatywy OR zmiennych START i SILNIK. Następnie, w zależności od uzyskanej wartości procedura JZ warunkowo może wykonać skok do etykiety `:?StartStop.MAIN?AND0003` lub przejść do wykonywania negacji NOT dla zmiennej STOP. Skok do tej etykiety implikuje, że finalnym wynikiem będzie ustawienie zmiennej SILNIK na 0 (`#00`). Ustawienie to poprzedza najpierw procedura MCD

zerująca zmienną tymczasową `?LR?ANDA0002` (linia 65), a potem skok JZ do etykiety `:?StartStop.MAIN?AND0001`, gdzie SILNIK jest zerowany (linia 73).



Rys. 2.4. Okno środowiska CPDev z kodem VMASM.

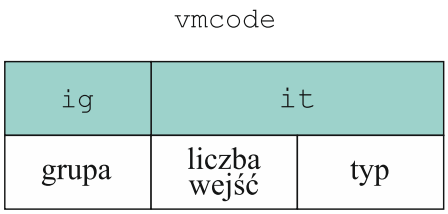
Natomiast w przypadku niezerowej wartości alternatywy OR, najpierw funkcja NOT w zmiennej `?LR?ANDA0006` określa wynik negacji STOP (linia 60). Zerowy wynik powoduje skok do etykiety `:?StartStop.MAIN?AND0003`, a dalej wyzerowanie zmiennej SILNIK jak poprzednio. W przeciwnym razie procedura MCD (linia 62) do zmiennej `?LR?ANDA0002` wpisuje wartość 1 (`#01`), po czym następuje skok JMP do etykiety `:?StartStop.MAIN?EAND0009`. Tam z kolei, wobec niezerowego `?LR?ANDA0002`, określana jest wartość negacji NOT wyjścia TP1.Q czasomierza i gdy ona również okaże się niezerowa, tzn. że minął czas zabezpieczenia T, zmienna SILNIK zostaje ustawiona na 1 (`#01`, linia 70).

2.4. Binarny kod wykonywalny

Generator kodu binarnego będący ostatnim elementem kompilatora (rys. 2.3) tworzy kod binarny poprzez asemblację programu w VMASM, zastępując mnemoniki instrukcji VMASM odpowiednimi identyfikatorami cyfrowymi, a operandy ich adresami w pamięci. W tym celu generator wykorzystuje plik konfiguracyjny LCF z definicjami instrukcji, w których ich mnemonikom (nazwom) odpowiadają identyfikatory cyfrowe [6,7].

Cyfrowy identyfikator instrukcji oznaczony jako `vmcode` jest dwubajtowym złożeniem identyfikatora grupy `ig`, do której należy instrukcja, oraz identyfikatora `it` konkretnego typu

danych. Strukturę vmcode pokazano na rysunku 2.5. W przypadku funkcji z przeciążonej grupy (możliwe różne typy), jak OR, NOT czy ADD, bajt ig oznacza rodzaj grupy, zaś it określa liczbę i typ operandów (wejść). Zmieniając it można wybierać konkretne funkcje z przeciążonej grupy. W ten sposób funkcje specyficzne dla typu, jak ADD:INT, AND:BOOL itp., mogą być wyrażone w formie cyfrowej. Przykładem vmcode może być 0514, oznaczający operację NOT (05) dla jednego wejścia (1) typu WORD (4). Gwiazdka (*) w polu liczby oznacza rozszerzalną liczbę wejść. Przykładowo, kod 09*0 reprezentuje operację OR (09), zwracającą wynik typu BOOL (0) dla liczby wejść określonej bezpośrednio w programie (*).



Rys. 2.5. Struktura identyfikatora instrukcji vmcode.

Definicje kilku funkcji i procedur w plikach źródłowych środowiska CPDev przedstawiono w tabeli 2.4 (operands i args są synonimami). Funkcje definiowane są jak wyżej, tzn. pierwszy bajt vmcode oznacza grupę, a drugi określa liczbę wejść i typ danych. Natomiast wszystkie procedury systemowe, jak JNZ, CALB, RETURN itp. należą do jednej grupy 1C. Druga połowa identyfikatora vmcode wskazuje wówczas na konkretną procedurę. Przykładowo, wywołania jednostek POU (*Program Organization Unit*), czyli programów oraz zdefiniowanych przez użytkownika bloków funkcyjnych i funkcji, wykonywane są przez parę procedur CALB, RETURN. Pojedyncze wywołanie ma postać CALB InstPtr, FunBlockAddr, gdzie InstPtr wskazuje instancję, a FunBlockAddr adres kodu POU (tutaj blok funkcyjny), np. w bibliotece (linia 54 na rys. 2.4).

Tab. 2.4. Przykładowe definicje funkcji i procedur.

Funkcja	**Procedura systemowa**

`<function name="ADD"`
`  vmcode="01*2"`
`  return="INT">`
`  <operands>`
`  <op no="*" name="a*"`
`  type="INT"/>`
`  </operands>`
`</function>`
	`<sysproc name="CALB"`
	`  vmcode="1C16">`
	`  <operands>`
	`  <op no="0" name="inst"`
	`  type=":rdlabel"/>`
	`  <op no="1" name="clbl"`
	`  type=":gclabel"/>`
	`  </operands>`
	`</sysproc>`

```
<function name="NOT"
          vmcode="0510"
          return="BOOL">
  <args>
    <arg no="0" name="INP"
         type="BOOL"/>
  </args>
</function>
```

```
<sysproc name="JNZ"
         vmcode="1C01">
  <operands>
    <op no="0" name="cnd"
        type="BOOL"/>
    <op no="1" name="clbl"
        type=":gclabel"/>
  </operands>
</sysproc>
```

```
<function name="EQ"
          vmcode="1204"
          return="BOOL">
  <args>
    <arg no="0" type="LINT"/>
    <arg no="1" type="LINT"/>
  </args>
</function>
```

```
<sysproc name="RETURN"
         vmcode="1C13">
  <args/>
</sysproc>
```

```
<function name="MIN"
          vmcode="0F05"
          return="BYTE">
  <args>
    <arg no="0" type="BYTE"/>
    <arg no="1" type="BYTE"/>
  </args>
</function>
```

```
<sysproc name="MCD"
         vmcode="1C15">
  <args>
    <arg no="0" name="DST"
         type=":rdlabel"/>
    <arg no="1" name="imm1"
         type=":imm.BYTE"/>
    <arg no="2" name="imm2"
         type=":imm.*"/>
  </args>
</sysproc>
```

Architektura maszyny wirtualnej, wyjaśniona szerzej w punkcie 3.1, implementuje kilka mechanizmów adresowania pamięci określanych przez typ atrybutu operandu w definicji instrukcji. Dostępne są następujące typy (dwa występują w procedurach powyżej):

- `:gclabel` – globalny wskaźnik (adres) w pamięci kodu,
- `:rclabel` – adres w pamięci kodu w stosunku do aktualnej zawartości rejestru kodu `CodeReg`,
- `:gdlabel` – globalny wskaźnik w pamięci danych,
- `:rdlabel` – adres w pamięci danych w stosunku do aktualnej zawartości rejestru danych `DataReg`,
- `:imm` – wartość natychmiastowa (bezpośrednia, stała).

Rejestr kodu `CodeReg` (licznik programu) jest zwiększany podczas pobierania argumentu instrukcji, z ewentualnymi zmianami wprowadzanymi przez skoki. Rejestr danych `DataReg` jest modyfikowany wyłącznie podczas wywoływania podprogramu, a po powrocie

przywracana jest jego poprzednia wartość. Przykładowy zapis `imm.WORD` wskazuje bezpośrednią wartość typu WORD.

Wygenerowanie końcowego kodu binarnego polega na konsolidacji pliku z mnemonikami VMASM, w którym wymieniane są:

- mnemoniki instrukcji na identyfikatory cyfrowe `vmcode`,
- nazwy zmiennych na adresy w pamięci danych,
- etykiety na adresy w pamięci kodu.

32-bitowy kod binarny dla sterowania silnikiem podano po lewej stronie rysunku 2.4 z odpowiadającymi mnemonikami VMASM po prawej. Pierwsza kolumna zawiera adres instrukcji (po dwukropku) w pamięci kodu, w drugiej znajduje się identyfikator `vmcode` a dalej adresy operandów i wartości bezpośrednie.

Różnice kodu binarnego dla wersji VM16 i VM32 objawiają się w rozmiarze etykiet i operandów, którymi są adresy, natomiast rozmiar identyfikatora `vmcode` pozostaje niezmieniony. Przykładowe 16- i 32-bitowe linie kodu mogą więc wyglądać następująco:

```
:00f1| 1C02 1E00 0D01
:00000167| 1C02 1E000000 93010000
```

Kod `1C02` jako `vmcode` oznacza procedurę skoku JNZ.

2.5. Informacje uzupełniające

Programy i biblioteki wchodzące w skład pakietu CPDev wymieniają dane poprzez pliki, których zawartość podano w tabeli 2.5.

Tab. 2.5. Podstawowe pliki pakietu CPDev.

Rozszerzenie	**Zawartość**
.xml	Źródłowy plik projektu
.cst	Kod programu w języku ST (plik tekstowy)
.vmasm	Kod mnemoniczny w języku VMASM
.dcp	Pośredni plik kompilatora z danymi zmiennych
.xcp	Kod binarny dla maszyny wirtualnej
.lcp	Prekompilowana biblioteka, np. IEC_61131
.scp	Dane sesji symulacyjnej
.xmc	Ustawienia komunikacyjne
.html	Raport projektu

Dane zmiennych w pliku *.dcp* zawierają nazwę, adres, typ adresu (lokalny, globalny), typ zmiennej i inne. Korzystają z nich symulator, konfigurator komunikacji [51], aplikacje Windows, a także inżynier wdrażający CPDev na danej platformie sprzętowej.

Szczególną aplikacją Windows, służącą przede wszystkim dydaktyce, jest tzw. CPCtrl (skrót od *Controller*) oznaczający maszynę wirtualną CPDev dla platformy x86 [52]. Funkcjonalnościami CPCtrl są:

- obsługa kart rozszerzeń I/O, jak National Instruments, Inteco,
- komunikacja Modbus TCP z systemem SCADA, np. AVEVA InTouch,
- serwer OPC dla aplikacji klienckich.

Z kolei WinController jest oprogramowaniem nadrzędnego komputera PC w systemie rozproszonym, wykonującego grupę powiązanych projektów CPDev za pomocą maszyn wirtualnych [37]. Każda maszyna może być odpowiednikiem CPCtrl. WinController jest usługą klasy *Windows service* funkcjonującą nieprzerwanie (uruchamianie automatyczne). Narzędzie WinController jest szerzej przedstawione w punkcie 6.3, służąc jako symulator wykonywania programów przez dwa rdzenie procesora.

Informacje techniczne dotyczące środowiska CPDev można również znaleźć na platformie GitHub [53].

3. Maszyna wirtualna

W rozdziale przedstawiono charakterystykę maszyny wirtualnej CPDev wykonującej kod binarny wygenerowany przez kompilator. Wprawdzie jest ona procesorem zrealizowanym programowo, ale podobnie jak typowy fizyczny procesor zawiera pamięci kodu i danych, stosy, rejestry, moduł przetwarzania instrukcji oraz interfejs dla docelowej platformy sprzętowej [7]. Algorytm modułu przetwarzania instrukcji przedstawiono zarówno w formie graficznej jak i w postaci modelu denotacyjnego [32,33], wskazującego wprost na rozwiązania programowe. Polega on na wyborze instrukcji do wykonania na podstawie identyfikatorów grupy `ig` *i typu* `it` *zawartych w identyfikatorze instrukcji* `vmcode`*. Podano także model denotacyjny oraz odpowiadającą mu realizację w C/C++ dla przykładowej przeciążonej i rozszerzalnej instrukcji ADD. Podstawowe elementy modeli denotacyjnych oraz następne przykłady znajdują się w Dodatku A. Symulację pracy maszyny w czasie rzeczywistym można przeprowadzić za pomocą symulatora CPSim [19]. Natomiast środowisko IDE, oprócz symulacji, udostępnia również funkcjonalności potrzebne do debugowania. Interfejs platformy docelowej maszyny bazowej składa się z prototypów funkcji niskopoziomowych, których treści odpowiadające konkretnym rozwiązaniom sprzętowym wypełniane są w procesie wdrażania [23].*

3.1. Architektura maszyny

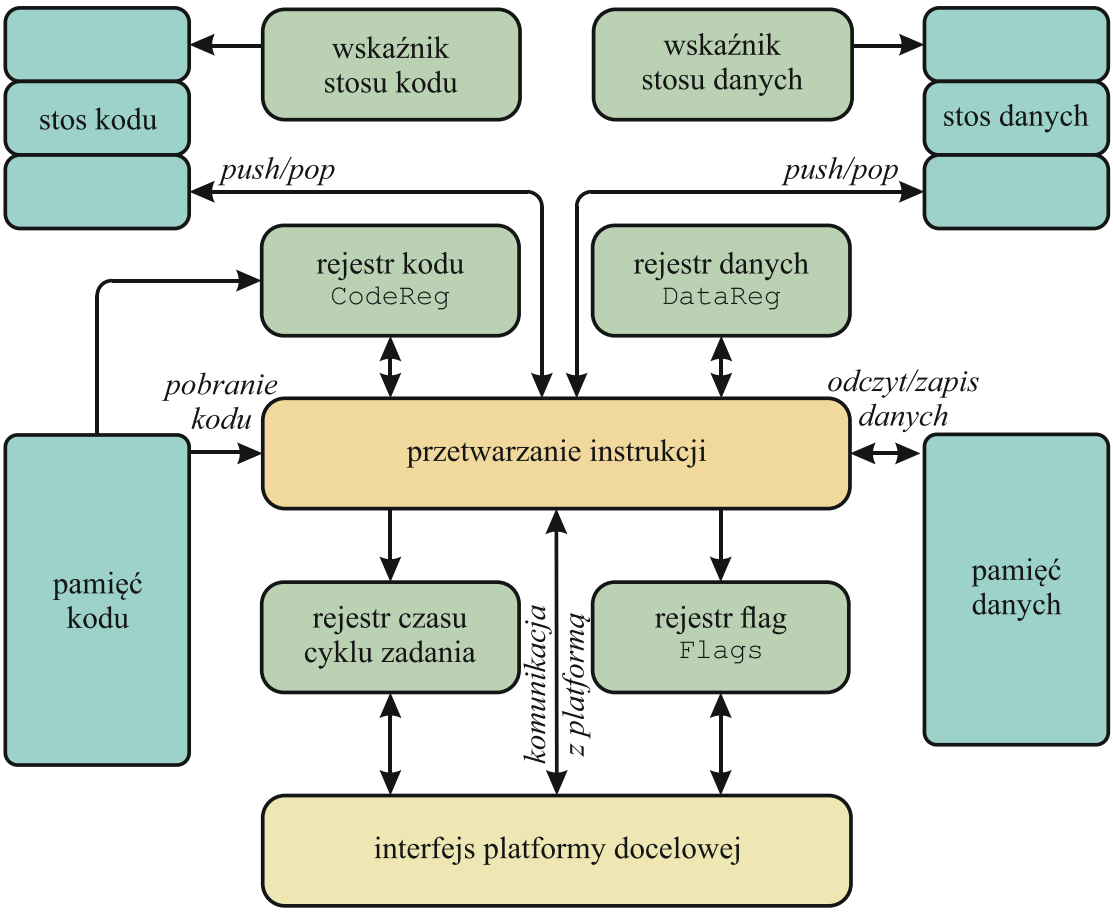
Elementy składowe maszyny wirtualnej CPDev oraz powiązania występujące między nimi pokazano na rysunku 3.1 [7]. Elementami tymi są:

- pamięci kodu i danych,
- stosy kodu i danych,
- rejestry i wskaźniki,
- moduł przetwarzania instrukcji,
- interfejs platformy docelowej.

Za interpretację kodu binarnego odpowiada moduł przetwarzania instrukcji. Pobiera on i wykonuje kolejne instrukcje z pamięci kodu, pobierając operandy (argumenty) z pamięci danych lub kodu. Wyniki wykonywania funkcji zapisywane są wyłącznie w pamięci danych. Jak wspominano wcześniej, maszyna wirtualna nie zawiera akumulatora ze względu na typy danych o różnych rozmiarach.

Rejestr kodu `CodeReg` reprezentuje licznik programu. Moduł przetwarzania instrukcji inkrementuje go każdorazowo po pobraniu instrukcji lub operandu. Rejestr danych `DataReg`

jest ustawiany poprzez wywołania i powroty z podprogramów, tzn. z bloków funkcyjnych i funkcji. Umożliwia to dostęp do zmiennych w różnych obszarach pamięci danych oraz obsługę instancji podprogramów. Podczas wywołania podprogramu bieżące wartości rejestrów `CodeReg` i `DataReg` są „wypychane” (*push*) na stosy kodu i danych. Po powrocie zawartości tych rejestrów są „ściągane” (*pull*) ze stosów. Mechanizm stosu zapewnia wielokrotne zagnieżdżanie bloków funkcyjnych. W maszynie wirtualnej znajduje się również rejestr `Flags` zawierający flagi statusowe sygnalizujące błędy lub nietypowe sytuacje, takie jak indeks tablicy poza zakresem, nieznany kod instrukcji, „zimny start” itp. Odpowiedni mechanizm platformy docelowej odczytuje aktualną zawartość `Flags` podejmując w razie potrzeby czynności zabezpieczające.



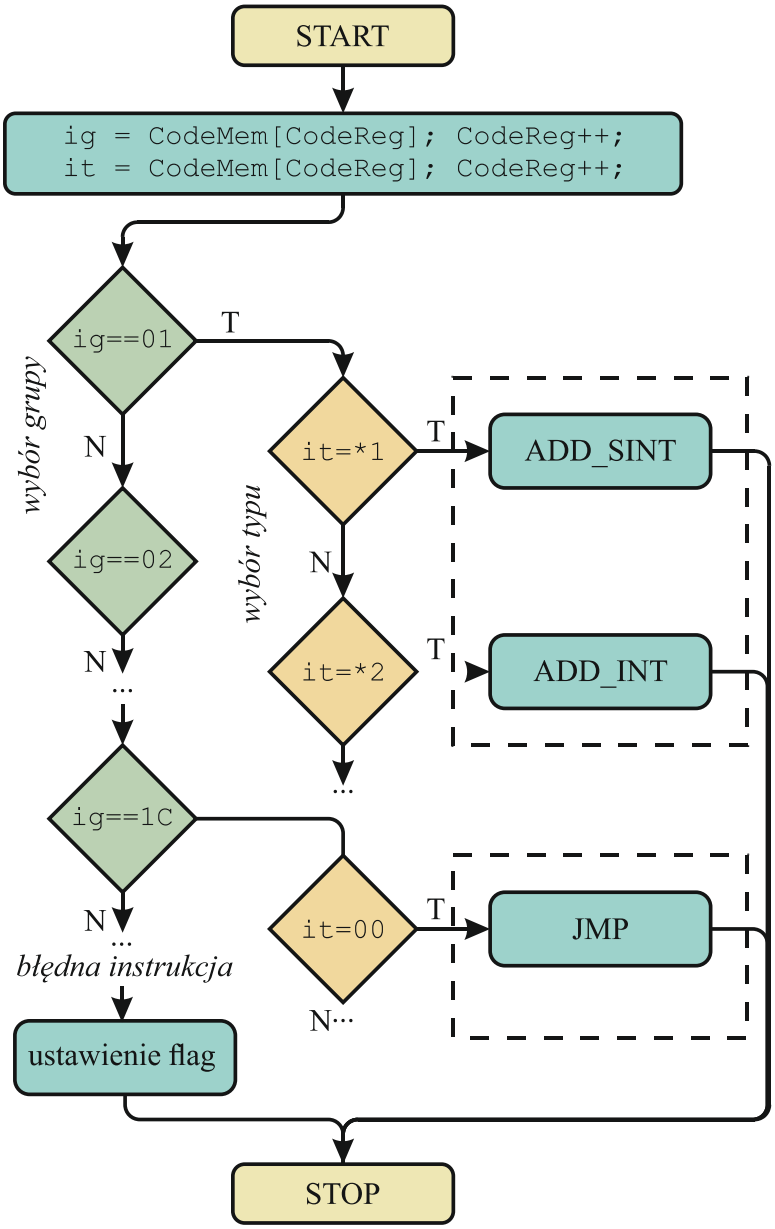
Rys. 3.1. Architektura maszyny wirtualnej.

Moduł interfejsu platformy docelowej udostępnia funkcje zależne od sprzętu i systemu operacyjnego, takie jak zarządzanie zegarem czasu rzeczywistego, utrzymywanie stałej wartości cyklu obliczeń, ładowanie programu, alokacja pamięci itp. Bazowa implementacja maszyny wirtualnej nie zawiera obsługi interakcji z otoczeniem zewnętrznym, czyli fizycznych wejść/wyjść lub komunikacji, ponieważ są one bezpośrednio zależne od sprzętu. Część ta

zarządzana jest przez oddzielny komponent oprogramowania implementowany przez inżyniera wdrażającego CPDev. Komponent ten wykorzystuje adresy zmiennych w pamięci danych nadawane w czasie kompilacji programu (plik *.dcp*). Środowisko dopuszcza również tzw. bloki natywne tworzone w C/C++ i wykorzystywane w analogiczny sposób jak standardowe bloki funkcyjne. Wykonywanie bloków natywnych jest znacznie szybsze niż bloków opracowanych w językach normy IEC 61131-3.

3.2. Algorytm przetwarzania instrukcji

Przetwarzanie instrukcji kodu binarnego składa się z kilku kroków, których graficzny algorytm przedstawiono na rysunku 3.2.



Rys. 3.2. Graficzny algorytm przetwarzania instrukcji.

Pierwszym krokiem jest pobranie identyfikatora instrukcji `vmcode` z pamięci kodu, będącego złożeniem bajtu `ig` oznaczającego grupę instrukcji oraz bajtu `it` określającego typ danych (rys. 2.5). Zakładając, że rejestr kodu `CodeReg` wskazuje na `vmcode`, najpierw pobierana jest wartość `ig` wraz z inkrementacją rejestru kodu. Potem następuje pobranie wartości `it` i następna inkrementacja. Dysponując wartościami `ig`, `it` algorytm dopasowuje identyfikator `ig` do konkretnej grupy funkcji lub procedur, po czym za pomocą `it` wybiera odpowiedni typ danych dla funkcji lub konkretną procedurę. Na tej podstawie wykonywany jest kod wybranej instrukcji. Implementacja algorytmu w maszynie wirtualnej opracowana w języku C opiera się o instrukcje wyboru `switch-case`.

3.3. Model denotacyjny

Algorytm graficzny pokazany wyżej można formalnie zapisać korzystając z semantyki denotacyjnej, pozwalającej definiować znaczenie programów za pomocą zrozumiałych modeli matematycznych [32,33]. Ponieważ modele te ogólnie przypominają programy, więc łatwiej jest na ich podstawie napisać program użytkowy w praktycznie dowolnym języku. W przeciwieństwie do modeli denotacyjnych, zoptymalizowane programy użytkowe zazwyczaj są trudne do analizy. Można więc uważać, że model denotacyjny jest pewnego rodzaju specyfikacją programu bazującego na architekturze maszyny.

Model denotacyjny maszyny wirtualnej CPDev został przedstawiony w [54,55], a jego zarys wraz z modelem powyższego algorytmu i przykładem modelu jednej instrukcji VMASM jest pokazany poniżej. Uzupełnienie modelu o funkcje niskopoziomowe oraz następne przykłady znajdują się w Dodatku A.

W ogólnym przypadku, nadrzędnym celem wykonania każdego programu jest wypracowanie nowego stanu komputera. Formalnym opisem stanu maszyny wirtualnej CPDev jest dziedzina $State$, będąca złożeniem dziedzin pamięci, stosów, rejestrów i flag, tzn.

$$State = CodeMemory \times DataMemory \times CodeStack \times DataStack \times \\ \times CodeReg \times DataReg \times Flags \tag{3.1}$$

Dziedzinę $State$ można również zapisać jako zbiór krotek

$$(cm, dm, cs, ds, cr, dr, flg) \tag{3.2}$$

zawierających konkretne wartości przypisane elementom maszyny. Jako komentarz można dodać, że w języku funkcyjnym F# [56] stan maszyny CPDev byłby zdefiniowany jako

```
type State =
   (Storage*Storage*Stack*Stack*Address*Address*Flags)
```

Koncepcję wykorzystania identyfikatora `vmcode` w modelu denotacyjnym oparto w [54,55] na uniwersalnej semantycznej funkcji $\mathcal{U}$ przetwarzającej aktualny stan maszyny na nowy, czyli

$$\mathcal{U}[\![any\_instruction]\!] = State \rightarrow State \tag{3.3}$$

gdzie $any\_instruction$ oznacza dowolny `vmcode`. Wewnętrznie, po zdekodowaniu identyfikatorów ig, it, funkcja $\mathcal{U}$ wywołuje wykonanie konkretnej instrukcji VMASM za pomocą funkcji

$$\mathcal{C}[\![instruction]\!] = State \rightarrow State \tag{3.4}$$

Model denotacyjny funkcji $\mathcal{U}$ wygląda więc następująco

$$\begin{array}{l}
\mathcal{U}[\![any_instruction]\!] = \lambda s. \\
(cm, dm, cs, ds, cr, dr, flg) \coloneqq s \\
ig \coloneqq Get1BMem(cr, cm) \\
cr_1 \coloneqq cr \oplus 1 \\
it \coloneqq Get1BMem(cr_1, cm) \\
cr_2 \coloneqq cr_1 \oplus 1 \\
s_1 \coloneqq \textbf{\textit{match}}\ ig\ \textbf{\textit{with}} \\
\quad \mid 01 \rightarrow \textbf{\textit{match}}\ it\ \textbf{\textit{with}} \\
\quad\quad \mid 22 \rightarrow \mathcal{C}{:}\,ADD{:}\,INT{:}\,r{:}\,op1{:}\,op2; \\
\quad\quad\quad\quad (cm, dm, cs, ds, cr_2, dr, flg) \\
\quad\quad \mid 32 \rightarrow \mathcal{C}{:}\,ADD{:}\,INT{:}\,r{:}\,op1{:}\,op2{:}\,op3; \\
\quad\quad\quad\quad (cm, dm, cs, ds, cr_2, dr, flg) \\
\quad\quad \mid \ldots \\
\quad\quad \textbf{\textit{end}} \\
\quad \mid \ldots \\
\quad \textbf{\textit{end}} \\
s_1
\end{array} \tag{3.5}$$

Model ma postać równania semantycznego (=) z tzw. wyrażeniem lambda [32,33] postaci $\lambda s.\, body$ gdzie s jest aktualnym stanem, a $body$ określa sekwencję operacji wytwarzających nowy stan s_1. Tutaj $body$ rozpoczyna się od unifikacji $(cm, dm, \ldots) \coloneqq s$ rozdzielającej stan s na krotkę z wartościami składników stanu. Zgodnie z algorytmem graficznym, rejestr kodu cr wskazuje początkowo na `vmcode`, który rozpoczyna obsługę pewnej instrukcji VMASM.

Dysponując wartościami składników stanu, funkcja $Get1BMem$ (Dodatek A) pobiera bajt identyfikatora grupy ig z pamięci kodu cm, którego adres znajduje się w rejestrze kodu cr. Towarzyszy temu inkrementacja tego rejestru do cr_1 (z dodawaniem $\oplus$ w ograniczonym zakresie – Dodatek A). Analogicznie pobierany jest identyfikator typu it, a rejestr kodu inkrementowany do cr_1. Następnie, po dwóch dopasowaniach ***match*** ... ***with***

z identyfikatorami ig, it, następuje wywołanie odpowiedniej funkcji $\mathcal{C}$ dla aktualnego stanu określonego przez krotkę $(cm, dm, cs, ds, cr_2, dr, flg)$.

W podanym w $\mathcal{U}$ przykładzie identyfikator $ig = 01$ reprezentuje dodawanie, a $it = 22{,}32$ dwa lub trzy operandy typu INT. W przypadku dwóch operandów jest to funkcja $\mathcal{C}$ [[`ADD:INT:r:op1:op2`]] z rezultatem r. Po wykonaniu wybranej funkcji $\mathcal{C}$ maszyna wirtualna znajduje się w nowym stanie s_1.

Model denotacyjny dodawania $\mathcal{C}$ [[`ADD…`]], również jako wyrażenie lambda, oraz jego implementacja w C/C++ są podane w tabeli 3.1. Po rozdzieleniu stanu s na krotkę, najpierw określany jest adres $raddr$ rezultatu r w pamięci kodu cm według zawartości rejestru kodu cr, z uwzględnieniem aktualnej zawartości rejestru danych dr (`ADD` jest funkcją). Inkrementacja cr do cr_1 zależy od $AddressSize$ (maszyna 16- i 32-bitowa). Podobnie określane są adresy operandów $op1addr$, $op2addr$ z rejestrem kodu inkrementowanym do cr_3. Odpowiednio do tych adresów funkcje $Get2BMem$ pobierają z pamięci danych dm pary bajtów i poprzez $IntOf$ konwertują je do typu INT. Suma $\oplus$ znajduje się w zmiennej sv, która po konwersji $FromInt$ aktualizuje poprzez $Upd2BMem$ pamięć danych w nowym stanie s_1.

Tab. 3.1. Model denotacyjny i implementacja C/C++ funkcji ADD.

Denotacja	**Implementacja w C/C++**

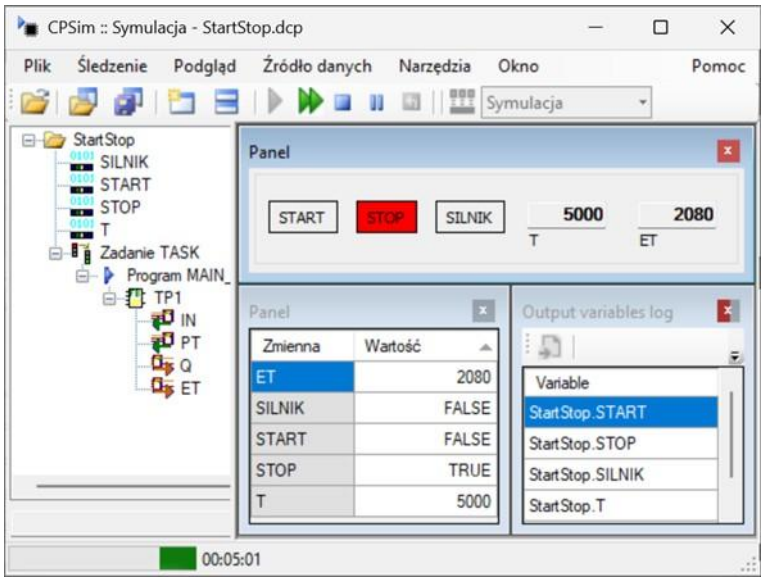
$\mathcal{C}[\![$`ADD:INT:r:op1:op2`$]\!] = \lambda s.$
$(cm, dm, cs, ds, cr, dr, flg) := s$
$r := GetAddress(cr, cm)$
$raddr := dr \oplus r$
$cr_1 := cr \oplus AddressSize$
$op1 := GetAddress(cr_1, cm)$
$op1addr := dr \oplus op1$
$cr_2 := cr1 \oplus AddressSize$
$op2 := GetAddress(cr_2, cm)$
$op2addr := dr \oplus op2$
$cr_3 := cr_2 \oplus AddressSize$
$sv := IntOf\big(Get2BMem(op1addr, dm)\big) \oplus$
$IntOf(Get2BMem(op2addr, dm))$
$s_1 := (cm, Upd2BMem(raddr, dm,$
$FromInt(sv)), cs, ds, cr_3, dr, flg)$
s_1
	`#define ADD_TYPE(TYPE)`
	`case IT_ADD_##TYPE & 0x000F: {`
	`  TYPE sv = 0;`
	`  BYTE i;`
	`  BYTE cnt = it >> 4;`
	`  ADDRESS raddr = dataReg +`
	`  GetCodeAddress();`
	`  for (i=0; i<cnt; i++) {`
	`  ADDRESS opxaddr = dataReg`
	`  + GetCodeAddress();`
	`  sv += TYPE##Of(GetMemData(`
	`  opxaddr,sizeof(TYPE)));`
	`  }`
	`  UpdMemData(raddr,`
	`  From##TYPE(sv),sizeof(sv));`
	`}`
	`break;`

Funkcja `ADD_TYPE` w implementacji C/C++ po prawej stronie tabeli 3.1 realizuje dodawanie dla wszystkich odpowiednich typów danych (przeciążenie) unikając w ten sposób powtarzania dość podobnego kodu. Rozpoznawanie właściwego typu następuje poprzez ostatni półbajt identyfikatora `vmcode` z maskowaniem za pomocą `0x000F` (druga linia). Wartość argumentu danego typu jest określana funkcją `TYPE##Of` na podstawie rozmiaru

`sizeof(TYPE)`. Ponadto podany kod obsługuje funkcje `ADD` z różną liczbą operandów (rozszerzalną), co reprezentuje `cnt` w pętli `for` (gwiazdka * w `it`, pkt. 2.4). Liczbę `cnt` uzyskuje się z `it` poprzez przesunięcie bitowe `>> 4` (pierwszy półbajt).

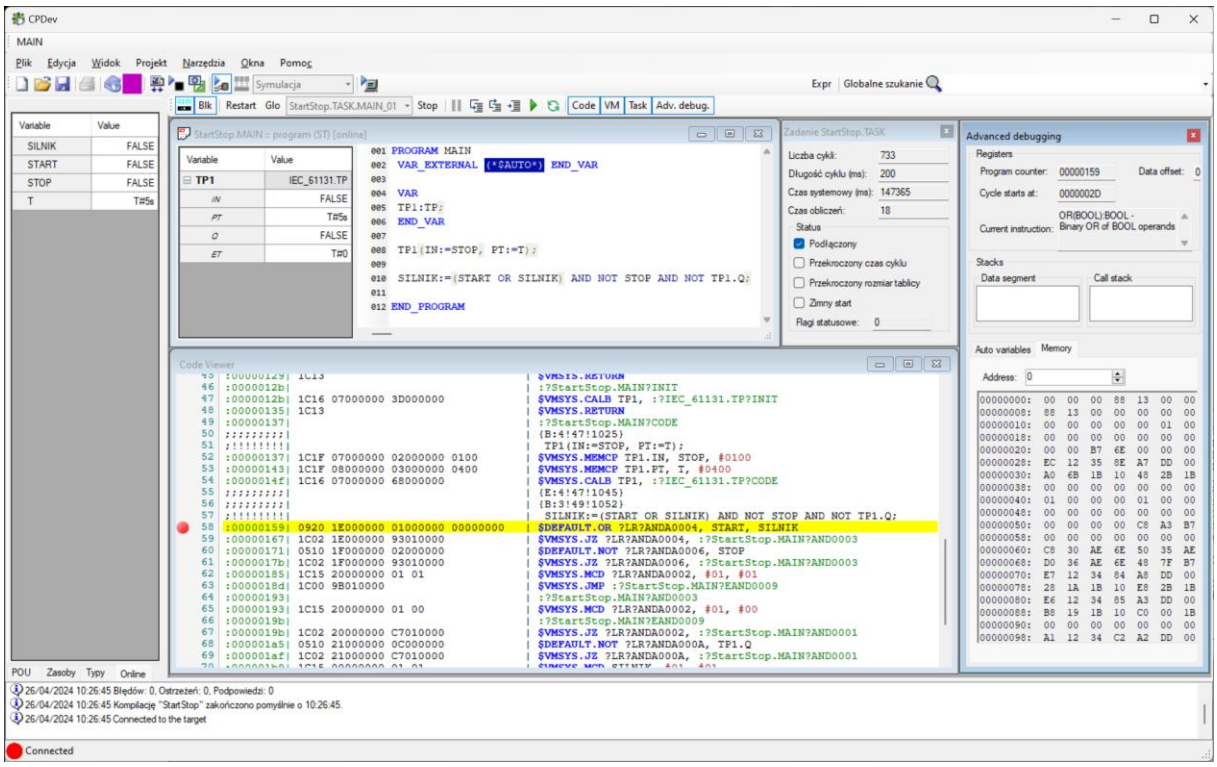
3.4. Symulacja i debugowanie

Od początku w skład środowiska CPDev wchodził symulator CPSim [19] służący do monitorowania działania programu sterowania w czasie rzeczywistym, ewentualnie także z dodatkowym programem symulującym obiekt. CPSim jest niezależną aplikacją Windows korzystającą z pliku *.dcp* generowanego przez kompilator (pkt. 2.5). Okno symulatora CPSim pokazano na rysunku 3.3 z drzewem projektu sterowania silnikiem po lewej stronie.



Rys. 3.3. Okno symulatora CPSim.

Symulator pozwala na przygotowanie paneli roboczych ułatwiających eksperymentowanie, które mogą zawierać przyciski, kontrolki LED, komórki wartości liczbowych albo listę zmiennych z aktualnymi wartościami. Sytuacja pokazana na rysunku reprezentuje zatrzymanie silnika przyciskiem STOP po 2080 ms od momentu załączenia. Inspiracją do opracowania CPSim było narzędzie PLCSIM firmy Siemens [57]. Wspierana jest również rejestracja wybranych zmiennych w pliku dziennika (*log*). Wykonywanie programu symulacyjnego można ewentualnie przyśpieszyć, gdy symulowany obiekt ma znaczną stałą czasową. Zastępując tryb *Symulacja* w pasku narzędziowym nazwą kanału komunikacyjnego, np. Modbus, uzyskuje się połączenie ze sterownikiem (tzw. *commissioning*). Źródłem danych dla CPSim może być również WinController [37].



Rys. 3.4. Środowisko CPDev IDE w trybie symulacji *online* i debugowania.

Debugowanie jest niezbędnym elementem pracy nad każdym programem, pozwalając na identyfikację i usuwanie błędów. Ma ono szczególne znaczenie w sterownikach i systemach sterowania ze względu na oddziaływanie na obiekt. Pakiet CPDev wspiera debugowanie w trybie *online* uruchamianym w bazowym środowisku IDE, czyli tam gdzie tworzony jest program. Podobnie jak CPSim, debugowanie zapewnia również połączenie z fizycznym sterownikiem poprzez wybrany sposób komunikacji.

Okna CPDev IDE składające się na debugowanie pokazano na rysunku 3.4. Pośrodku w oknie *StartStop.MAIN::program (ST) [online]* jest widoczny wykonywany program źródłowy z wartościami wejść i wyjść bloku TP1 zdefiniowanego lokalnie. Po lewej stronie znajduje się okno z wartościami zmiennych globalnych umożliwiające interakcję z programem, tzn. ustawianie wejść START, STOP. Okno *CodeViewer* zawiera kod binarny i mnemoniki języka VMASM (jak rys. 2.4), przy czym podświetlona linia oznacza punkt wstrzymania wykonywania kodu (*breakpoint*). Wykonywanie kodu można kontrolować za pomocą przycisków w górnym pasku narzędziowym. Górna część okna *Advanced debugging* po prawej stronie pokazuje aktualny stan maszyny wirtualnej, a dolna zawartość pamięci danych. Dodatkowe informacje dotyczące zadania, jak liczba cykli, długość cyklu itp. znajdują się w oknie *Zadanie*. Wymienione funkcjonalności debugowania w CPDev IDE są typowe także dla innych środowisk programistycznych.

3.5. Interfejs platformy docelowej

Jak wcześniej podkreślano, unikalną cechą pakietu CPDev jest możliwość dostosowania go do sprzętu konkretnego sterownika. Oznacza to możliwość wyboru jednostki centralnej, na której uruchamia się maszynę wirtualną, a także możliwość obsługi różnych dodatkowych urządzeń peryferyjnych. CPDev jest także otwarty na specyficzne rozwiązania elektroniczne i informatyczne, także takie, które rzadko spotyka się w automatyce przemysłowej.

Elementem wiążącym maszynę wirtualną z rozwiązaniami sprzętowymi sterownika jest interfejs platformy docelowej (rys. 3.1), którego zasadniczą częścią są funkcje niskiego poziomu wywoływane przez maszynę. Specyfikacje tych funkcji wyrażone są poprzez prototypy (puste) niezależne od procesora i rozwiązania sprzętowego, o nazwach rozpoczynających się od `VMP` (*Virtual Machine Platform*). Podstawowy zestaw prototypów przedstawiono w [23] na podstawie wcześniejszych doświadczeń z urządzeniami wielofunkcyjnymi [58]. Główne prototypy są wymienione poniżej.

- `VMP_LoadConfiguration` – ładuje parametry zadania (cykl, liczba i kolejność POU), kod binarny oraz przydziela pamięć na dane; rozmiar pamięci jest określany przez kompilator według pliku LCF (tab. 2.3),
- `VMP_PreRunConfiguration` – inicjuje peryferia i przechowuje stan początkowy (czas, parametry),
- `VMP_PostRunConfiguration` – zwalnia zasoby wykorzystywane przez zadanie,
- `VMP_PreCycle` – aktualizuje zmienne programu przed obliczeniami na podstawie wejść, przechowuje stan początkowy zegara systemowego,
- `VMP_PostCycle` – aktualizuje wyjścia na podstawie wyników obliczeń; jeśli pozostał czas do końca cyklu, to uruchamia testy lub inne zadania zależne od platformy,
- `VMP_CurrentTime` – zwraca aktualną wartość zegara systemowego dla zmiennych typu TIME,
- `VMP_ReadRTC` – zwraca odczyt zegara czasu rzeczywistego dla zmiennych typu DATE_AND_TIME.

W przypadku przekroczenia czasu cyklu funkcja `VMP_PostCycle` ustawia odpowiedni bit w rejestrze `Flags` maszyny wirtualnej (rys. 3.1). Na tej podstawie interfejs platformy docelowej podejmuje określone kroki, np. ustawienie bezpiecznych wyjść. Z założenia kod wypełniający prototypy `VMP` jest tworzony w języku C/C++ przez inżyniera wdrażającego CPDev na docelowej platformie i konsolidowany z bazową maszyną wirtualną.

W maszynie wirtualnej, podobnie jak w typowych sterownikach, podczas wykonywania pojedynczego cyklu programu stosuje się semantykę odczyt-wykonanie-zapis. Oznacza to, że na początku cyklu (*precycle*) odczytywane wartości wejść są zapisywane do wewnętrznych kopii lokalnych. W głównej części cyklu, czyli podczas wykonywania programu sterowania, używane są wyłącznie zapisane kopie. Mechanizm ten czasami nazywany jest obrazem procesu (*process image*). Ostatnim etapem cyklu jest aktualizacja wartości wyjść (*postcycle*), gdzie stan obliczonych kopii lokalnych przenoszony jest do globalnych zasobów wykorzystywanych w następnym cyklu. Kompilator realizuje mechanizm obrazu procesu za pomocą zestawu instrukcji kopiowania MEMCP (tab. 2.2), które przenoszą wartości zmiennych globalnych do kopii i odwrotnie.

4. Zwiększanie wydajności metodami dostępu do pamięci

Rozdział przedstawia metody dostępu do pamięci z perspektywy maszyny wirtualnej CPDev. Występujące ograniczenia spotykane w różnych architekturach procesorów wskazały na możliwość zwiększania wydajności poprzez wybór właściwiej metody dostępu do pamięci. Metody BYTE_ACCESS i MEMCPY_ACCESS bazują na podstawowej wersji maszyny wirtualnej. Trzecia metoda DIRECT_ACCESS skierowana jest do jednostek 32-bitowych, najlepiej w połączeniu z trybem wyrównania do czterech bajtów ALIGN_4B. Przedstawione metody dostępu i tryby adresacji 16-, 32-bitowa i 32-bitowa z wyrównaniem charakteryzują się zróżnicowaną wydajnością oraz zajętością pamięci. Ostateczny wybór konkretnego rozwiązania powinien zależeć od platform docelowych oraz wymogów projektowych.

4.1. Architektury ARM

Obecnie 70% systemów wbudowanych wykorzystuje procesory oparte na architekturze ARM (*Advanced RISC Machine*) [59]. Masowe wdrożenia ARM są wynikiem polityki licencyjnej przekazującej etap tworzenia i produkcji własnych układów firmom zewnętrznym. Czołowi producenci to STMicroelectronics, Raspberry Pi Foundation, NXP Semiconductors, Texas Instruments i inni.

Pierwszy rdzeń ARM bazujący na architekturze ARMv1 wprowadzono w 1985 roku. W kolejnych latach powstały nowe konstrukcje o zmodyfikowanych architekturach [27]. Aktualnie zakres ten obejmuje rdzenie: Cortex serii M – mikrokontrolery, A – profil ogólny, aplikacyjny, R – optymalizacja pod kątem czasu rzeczywistego i bezpieczeństwa, oraz Neoverse – rdzeń serwerowy. W przeciwieństwie do A i R, rdzeń M nie zawiera jednostki zarządzania pamięcią MMU (*Memory Management Unit*), która jest kluczowym modułem do uruchamiania systemów operacyjnych, takich jak Linux.

Rdzenie Cortex-M i Cortex-A zyskały szczególne uznanie producentów i użytkowników. Linia M została zoptymalizowana pod kątem budowy mikrokontrolerów kładąc nacisk na wysoką efektywność energetyczną i niski koszt produkcji. Charakteryzuje ją szerokie spektrum zastosowań, poczynając od niskowydajnych i ekonomicznych, po energooszczędne lub wysoce wydajne jednostki obliczeniowe do zadań specjalnych, jak mechatronika, sztuczna inteligencja, szyfrowanie, czy renderowanie obrazu. Z kolei architektura Cortex-A obejmuje procesory z wydajnymi 32- i 64-bitowymi rdzeniami, wyposażonymi w jednostki MMU. Jej głównymi zastosowaniami są urządzenia przenośne

wyposażone w systemy operacyjne, czyli smartfony, komputery jednopłytkowe, routery, kamery itd.

Nacisk na efektywność energetyczną i niski stopień komplikacji sprawił, że układy ARM mają określone ograniczenia. Część architektur, zwłaszcza wcześniejszych, charakteryzuje się ograniczeniem zwanym adresacją wyrównaną [27]. Dostęp wyrównany oznacza taką operację odczytu lub zapisu, w której adres danej jest podzielny przez liczbę bajtów określających jej rozmiar. Dla słowa 4-bajtowego oznacza to podzielność adresu przez cztery, a dla półsłowa (2B) podzielność przez dwa. Dostęp do bajtów nie wymaga wyrównania.

Przykładowo, podstawowe serie mikrokontrolerów STM32 firmy STMicroelectronics jak F0 i G0, oparte zostały odpowiednio na rdzeniach Cortex-M0 i M0+, które nie wspierają operacji na niewyrównanych danych. Każdorazowa próba takiego dostępu powoduje wywołanie krytycznego wyjątku (*Alignment Fault*) i przejście w tryb jego obsługi (*Hard Fault Handler*). Obsługa zależy od aplikacji, zazwyczaj jest to zablokowanie programu poprzez wykonywanie nieskończonej pętli aż do resetu, z opcjonalnym logowaniem danych diagnostycznych. W niektórych wersjach architektury operacje na adresach niepodzielnych przez rozmiar zmiennej są dozwolone, lecz działania na niewyrównanych danych zmniejszają wydajność aplikacji i zwiększają zużycie energii. Omówiono to bliżej w punkcie 4.3.

Zróżnicowanie architektur procesorów przyczynia się do skomplikowania procesu wdrażania oprogramowania. Problem ten występuje nawet w przypadku jednego producenta. Przykładowo, w rodzinie mikrokontrolerów STM32 układy są wyposażone w następujące rdzenie Cortex-M:

- M0/M0+ – ARMv6-M, Von Neumann,
- M3 – ARMv7-M, Harvard,
- M4/M7 – ARMv7E-M, Harvard.

Architektura ARM może więc bazować zarówno na koncepcji Von Neumanna korzystającej z jednej pamięci i magistrali do obsługi danych i instrukcji, albo na strukturze typu Harvard oddzielającej te zasoby.

Pomimo dominacji architektury ARM, dostępny jest również szereg innych rozwiązań dla systemów wbudowanych. Architektury Xtensa LX i Diamond 106Micro oparte na Xtensa ISA (*Xtensa Instruction Set Architecture*) [60] występują w popularnych układach ESP32 i ESP8266. Architektura MIPS wprowadzona w 1985 roku składa się z kilku generacji: MIPS I do MIPS V [21], MIPS32 oraz microMIPS. W 2021 roku jej rozwój został jednak ograniczony na rzecz RISC-V [61]. W 2013 roku Intel wprowadził 32-bitową serię procesorów Quark o niskim zużyciu energii i niewielkich rozmiarach, obsługującą podstawowy zestaw instrukcji

x86 [62]. Porównanie wydajności wymienionych a także innych architektur zastosowanych w konkretnych układach jako maszyna wirtualna CPDev zostało opisane w [48]. Wyniki badań wydajności dla wyrównanych i niewyrównanych danych można również znaleźć w [63,64].

4.2. Dostęp do pamięci metodami BYTE_ACCESS i MEMCPY_ACCESS

Dostęp do pamięci może wiązać się z ograniczeniami sposobu ulokowania danych o różnych rozmiarach. W kontekście wyrównywania zmiennych, operacje na pojedynczych bajtach obsługiwane są przez większość architektur. Dla uzyskania kompatybilności maszyny wirtualnej z ograniczeniami dostępu do pamięci, została opracowana metoda BYTE_ACCESS polegająca na składaniu rozszerzonych typów danych, tzn. 2-, 4-, 8-bajtowych, z pojedynczo odczytanych bajtów, podobnie jak to było w pierwszej wersji maszyny CPDev [19]. Przykładowo, odczyt słowa (4B) wymaga dwóch operacji odczytu półsłów (czterech operacji odczytu bajtu) oraz przesunięcia i złożenia bitowego. Metoda ta jest jednym z rozwiązań przy zmianie kolejności bajtów (*endianness*), np. z *Little-Endian* na *Big-Endian* lub odwrotnie. Warto dodać, że niektóre architektury ARM umożliwiają dynamiczne przełączanie między tymi trybami bez modyfikacji kodu (*Big-Endian Execution Mode*). Wadą metody BYTE_ACCESS jest zwiększony czas potrzebny na odczytywanie pojedynczych bajtów, a następnie na ich składanie. Uproszczoną implementację dla półsłów WORD i słów DWORD w języku C przedstawiono na listingu 4.1.

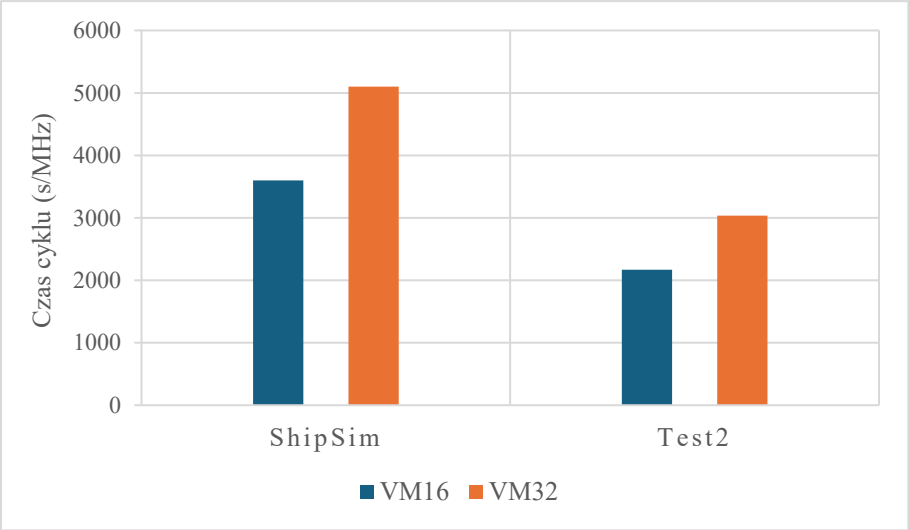
```
WM_WORD GetProgramWord(){
  WM_WORD lo = GetProgramByte();
  WM_WORD hi = GetProgramByte();
  return (hi<<8) | lo;}

WM_DWORD GetProgramDWord(){
  WM_DWORD lo = GetProgramWord();
  WM_DWORD hi = GetProgramWord();
  return (hi<<16) | lo;}
```

Listing 4.1. Implementacja instrukcji pobierania zmiennych typu WORD oraz DWORD w trybie BYTE_ACCESS.

Dla oceny metod dostępu do pamięci przygotowano cztery testy opisane w [34], które wykorzystują różnorodne operacje maszyny wirtualnej. Są to: symulator statku – ShipSim, obliczanie liczb doskonałych – Test1, poszukiwanie liczb pierwszych – Test2, konwersja dziesiętne/binarne – Test3. Ponieważ wyniki Testów 1,2,3 nie różnią się zasadniczo, poniżej przedstawiono je tylko w odniesieniu do ShipSim i Test2. Dla wyjaśnienia, ShipSim jest rozbudowanym programem wykorzystującym zmienne typu REAL, będącym symulatorem

statku morskiego o zadanych parametrach [65]. Test2 poszukuje liczb pierwszych wykorzystując operacje arytmetyczne modulo, dzielenie, porównanie, a także przesunięcie i operacje bitowe. Metoda BYTE_ACCESS została przetestowana w 16- i 32-bitowym trybie pracy maszyny wirtualnej oznaczonym poprzednio jako VM16 i VM32. Porównanie względnego średniego czasu cyklu dla testów ShipSim oraz Test2 przedstawiono na rysunku 4.1. Wartości podane są w sekundach na 1 MHz częstotliwości zegarowej procesora, przy czym dla ShipSim zwiększono je 5000 razy celem wyrównania zakresów ze względu na czasochłonny algorytm poszukiwania liczb pierwszych.

Charakterystyczną cechą metody BYTE_ACCESS jest niewykorzystywanie sprzętowych rozwiązań przyśpieszających dostęp do rozszerzonych typów danych. W wyniku tego, czasy wykonywania cyklu maszyny wirtualnej zwiększyły się o około 40% na niekorzyść maszyny VM32. Tak więc, gdy rozmiary pamięci kodu i danych nie przekraczają 64 kB, metoda BYTE_ACCESS wyraźnie preferuje maszynę 16-bitową.



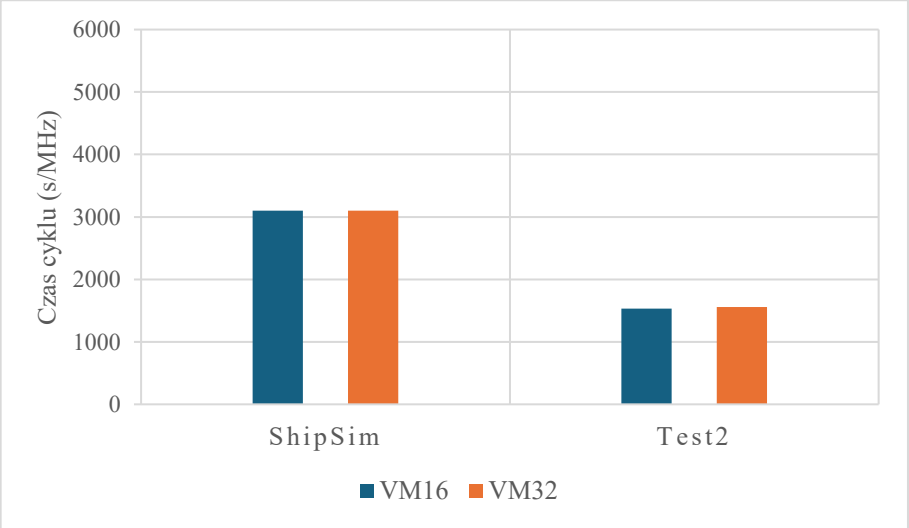
Rys. 4.1. Względne średnie czasy wykonywania cykli programów testowych na maszynie 16- i 32-bitowej w trybie BYTE_ACCESS (Cortex-M4).

Alternatywna metoda MEMCPY_ACCESS polega na kopiowaniu komórek pamięci przy użyciu standardowej funkcji `memcpy` dostępnej w językach C, C++ i in. Jej wykorzystanie powoduje przerzucenie wyboru optymalnej metody kopiowania na kompilator i bibliotekę dostosowaną do specyfiki danej platformy sprzętowej. Metoda jest uniwersalna i elastyczna pod względem możliwości zastosowania w różnych procesorach. Uproszczoną implementację dla WORD i DWORD w C przedstawiono na listingu 4.2, gdzie argumentami `memcpy` są wskaźniki do obiektu docelowego i źródłowego oraz rozmiar pamięci podlegającej kopiowaniu.

```
WM_WORD GetProgramWord(){
  WM_WORD wRes;
  memcpy(&wRes, getCodePtr(wProgramCnt), sizeof(WM_WORD));}

WM_DWORD GetProgramDWord(){
  WM_DWORD wRes;
  memcpy(&wRes, getCodePtr(wProgramCnt), sizeof(WM_DWORD));}
```

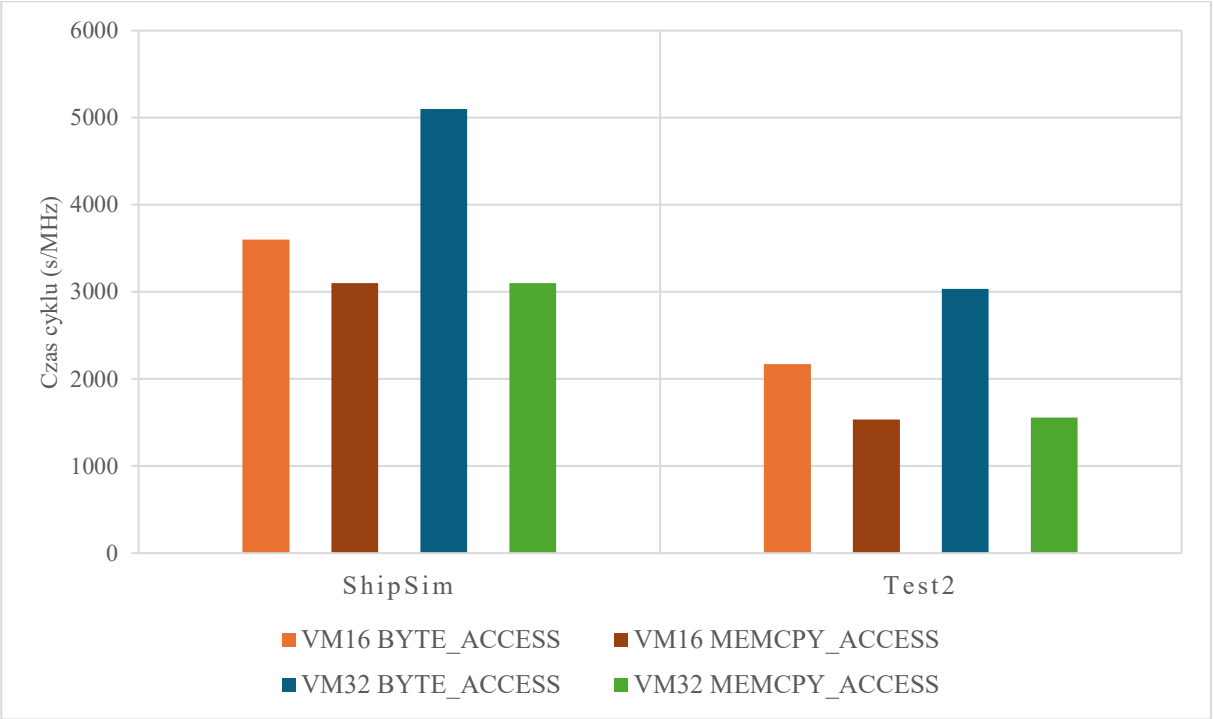
Listing 4.2. Implementacja instrukcji pobierania zmiennych typu WORD oraz DWORD w trybie MEMCPY_ACCESS.

Metodę MEMCPY_ACCESS przetestowano jak poprzednio uzyskując wyniki pokazane na rysunku 4.2. Szczególną korzyścią z wykorzystania funkcji `memcpy` jest zminimalizowanie różnicy czasu cyklu sterownika między 16- i 32-bitowym trybem pracy, która okazała się nie większa niż 2.5%. Jest to więc rozwiązanie uniwersalne dla różnych architektur, szczególnie z ograniczeniami dostępu do pamięci.



Rys. 4.2. Względne średnie czasy wykonywania cykli programów testowych na maszynie 16- i 32-bitowej w trybie MEMCPY_ACCESS (Cortex-M4).

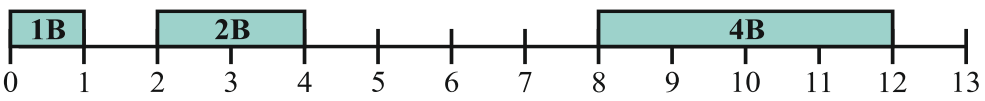
Finalne porównanie wyników obydwu metod dostępu dla 16- i 32-bitowej maszyny wirtualnej przedstawiono na rysunku 4.3. Dla 16-bitowej wersji metoda MEMCPY_ACCESS przyczyniła się do zmniejszenia czasu cyklu dla ShipSim i Test2 odpowiednio o 16% i 41% w stosunku do BYTE_ACCESS. Dla wersji 32-bitowej różnica jest znacznie większa i wynosi 64% i 94%. Mając więc na względzie wyłącznie wydajność, metoda MEMCPY_ACCESS uzyskała każdorazowo lepsze wyniki niż BYTE_ACCESS. Pozwala to zatem na stosowanie bez utraty wydajności rozbudowanej wersji maszyny VM32 zamiast VM16, niezależnie od rozmiaru pamięci programu i danych.



Rys. 4.3. Względne średnie czasy wykonywania cykli programów testowych na maszynie 16- i 32-bitowej w trybach BYTE_ACCESS i MEMCPY_ACCESS (Cortex-M4).

4.3. Metoda DIRECT_ACCESS z wyrównaniem danych ALIGN_4B

Operacje na pamięci za pomocą wskaźników są bezpośrednią metodą zapisu i odczytu danych. Zakodowane w języku C pod znakami * i & zlecają kompilatorowi wykonanie ściśle określonych poleceń, co jednak w przypadku próby odczytu nieodpowiednio ulokowanych danych może zakończyć się krytycznym wyjątkiem zgłaszanym przez procesor. Operacje na niewyrównanych danych mogą także generować kary czasowe. Choć zazwyczaj ograniczenia te nie są brane pod uwagę przez programistę ze względu na optymalizację kompilatora, to w przypadku kodu pośredniego dla maszyn wirtualnych należy uwzględnić właściwości platformy docelowej. Przykładowy sposób wyrównanego lokowania różnych rozmiarów danych przedstawiono na rysunek 4.4 [48]. Adresy zmiennych 1B są dowolne, adresy zmiennych 2B wymagają podzielności przez 2 (0, 2, 4,…), a zmiennych 4B podzielności przez 4 (0, 4, 8, …). Rozkład danych w pamięci może być optymalizowany przez kompilator pod kątem zajętości niwelując puste pola.



Rys. 4.4. Przykładowe wyrównane lokowanie danych o różnych rozmiarach.

Architektura harwardzka maszyny wirtualnej wyróżnia dwa obszary pamięci, które należało dostosować do występujących ograniczeń. W pamięci danych zmienne rozkładane są zgodnie z adresacją określoną przy ich definiowaniu, natomiast rozkład pamięci kodu zależy od kompilatora języka ST zastosowanego w CPDev. Jego przystosowanie do wyrównywania danych do 4B wymagało:

- wprowadzenia dodatkowych procedur systemowych,
- generowania kodu pośredniego z instrukcjami wyrównanymi w pamięci kodu.

Metoda DIRECT_ACCESS operuje bezpośrednio na pamięci układanej w ściśle uporządkowany sposób oznaczony ALIGN_4B (*4 Byte Alignment*) zwiększając tym samym liczbę kompatybilnych architektur. Opracowane dla tej metody nowe warianty procedur inicjujących, kopiujących i wypełniających dane z wyrównaniem zebrano w tabeli 4.1. Zmodyfikowane wersje otrzymały dodatkowe oznaczenie 4B. Zatem w trybie ALIGN_4B zamiast podstawowego wypełniania `FPAT` używana jest instrukcja `FPAT4B`. Dzięki wprowadzonym zmianom zmieniły się również możliwości instrukcji, bo np. procedura `MCD4B` umożliwia inicjowanie obszarów pamięci danych o rozmiarze nawet do 4 GB (wcześniej 255 bajtów). Ostatnie instrukcje `HWFBI4B`, `HWFBC4B` w tabeli 4.1 pozwalają na przekazywanie danych z kodu ST do bloków natywnych zależnych od platformy docelowej.

Tab. 4.1. Procedury systemowe dla trybu ALIGN_4B.

Procedura systemowa	**Argumenty**	**Opis**
`MCD4B`	`dst, size, pattern`	Inicjacja pamięci danych od adresu `dst` danymi z pamięci kodu `pattern` o rozmiarze `size` (wyrównanie do 4B za pomocą 0xFF)
`MEMCP4B`	`dst, src, count`	Kopiowanie obszaru pamięci danych z `src` do `dst` o rozmiarze `count` z wyrównaniem
`FPAT4B`	`dst, val, count`	Wypełnianie obszaru `dst` o rozmiarze `count` wartością `val` z wyrównaniem
`DPRDL4B`	`var, src, count`	Kopiowanie danych z `src` o rozmiarze `count` do zmiennej `var`
`DPWRL4B`	`var, dst, count`	Kopiowanie wartości zmiennej `var` do obszaru `dst` o rozmiarze `count`
`HWFBI4B` `HWFBC4B`	`id, instance`	Inicjacja i wywołanie instancji `instance` bloku natywnego typu `id`

Podstawowa procedura `MCD` z tabeli 2.2 w rozdziale 2 kopiuje obszar pamięci kodu do pamięci danych. Pozwala to na zainicjowanie tablicy w pamięci danych wartościami zadeklarowanymi w kodzie, umieszczonymi przez kompilator po dwóch pierwszych

operandach. Jej rozszerzenie `MCD4B` działające w trybie ALIGN_4B dodatkowo wyrównuje dane do 4B za pomocą ustawiania dodatkowych bajtów (0xFF).

Dla porównania `MCD` z `MCD4B`, w tabeli 4.2 podano ich modele denotacyjne stosując oznaczenia pamięci, stosów, rejestrów i flag jak w punkcie 3.3. Pierwszy operand `dst` jest etykietą docelową w pamięci danych, drugi `size` jest wartością bezpośrednią wskazującą liczbę bajtów do inicjacji, a trzeci `pattern` oznacza dane źródłowe w pamięci kodu. Pierwszym krokiem w obydwu procedurach jest wyznaczenie fizycznego adresu $dstaddr$ w pamięci danych podlegającego inicjacji uwzględniając aktualną zawartość rejestru danych dr. W następnym kroku rozmiar kopiowanych danych jako cr_1 jest pobierany z rejestru kodu. W zależności od procedury, jest to wykonywane w `MCD` za pomocą funkcji $Get1BMem$ i $ByteOf$ lub $Get4BMem$ i $DWordOf$ w `MCD4B`. Wynik zostaje zapisany w zmiennej $size$. Wykonanie powyższych operacji wymaga aktualizacji rejestru kodu do wartości cr_2 wskazującej na obszar kopiowania. Przesunięcie rejestru kodu następuje o 1 lub 4 bajty zależnie od procedury. Kopiowanie pamięci realizuje funkcja $MemMove$. Dodatkowo, w `MCD4B` przeprowadzane jest dopasowanie nowej wartości cr_4 rejestru kodu dla wyrównania do 4 bajtów (podzielność adresu przez 4). Wynikiem działania procedury jest nowy stan s_1 maszyny wirtualnej, w którym oprócz zaktualizowanej pamięci danych um występuje zaktualizowana wartość rejestru kodu, tj. adresu następnej instrukcji po skopiowanym obszarze. Pozostałe procedury z tabeli 4.1 zmodyfikowano w analogiczny sposób uzyskując kompatybilność kodu binarnego z procesorami niewspierającymi dostępu niewyrównanego.

Tab. 4.2. Modele denotacyjne procedur systemowych inicjujących dane – `MCD` (bez wyrównania) i `MCD4B` (z wyrównaniem).

$C[\![\texttt{MCD:dst:size:pattern}]\!] = \lambda s.$ $(cm, dm, cs, ds, cr, dr, flg) \coloneqq s$ $dst \coloneqq GetAddress(dr, cm)$ $dstaddr \coloneqq dr \oplus dst$ $cr_1 \coloneqq cr \oplus AddressSize$ $size \coloneqq ByteOf(Get1BMem(cr_1, cm))$ $cr_2 \coloneqq cr_1 \oplus 1$ $um \coloneqq MemMove(dstaddr,$ $dm, cr_2, cm, size)$ $s_1 \coloneqq (cm, um, cs, ds, cr_2 \oplus \text{size}, dr, flg)$ s_1	$C[\![\texttt{MCD4B:dst:size:pattern}]\!] = \lambda s.$ $(cm, dm, cs, ds, cr, dr, flg) \coloneqq s$ $dst \coloneqq GetAddress(dr, cm)$ $dstaddr \coloneqq dr \oplus dst$ $cr_1 \coloneqq cr \oplus AddressSize$ $size \coloneqq DwordOf(Get4BMem(cr_1, cm))$ $cr_2 \coloneqq cr_1 \oplus 4$ $um \coloneqq MemMove(dstaddr,$ $dm, cr_2, cm, size)$ $cr_3 \coloneqq cr_2 \oplus size \oslash 4$ $cr_4 \coloneqq cr_3 \oplus \textbf{match}\ size\ mod\ 4\ \textbf{with}$ $\mid 0 \rightarrow 0$ $\mid \_ \rightarrow 4$ $s_1 \coloneqq (cm, um, cs, ds, cr_4, dr, flg)$ s_1

Na listingu 4.3 przedstawiono fragment kodu VMASM skompilowanego w trybie ALIGN_4B. Instrukcje w liniach od drugiej do czwartej zawierają nowe procedury `FPAT4B` i `MCD4B` inicjujące tablice `BUF1`, `BUF2` oraz zmienną `I` zadanymi wartościami odpowiednio do operandów z tabeli 4.1. W ostatniej instrukcji widoczne jest nadmiarowe wypełnienie obszaru danych wartościami 0xFF w celu wyrównania do 4B.

```
:00000020| :?CpyMem.CPYMEM_BYTE?INIT
:00000020| $VMSYS.FPAT4B BUF1, #0000, #1400
:0000002c| $VMSYS.FPAT4B BUF2, #0000, #1400
:00000038| $VMSYS.MCD4B I, #02000000, #0000FFFFFFFF
```

Listing 4.3. Fragment kodu VMASM dla trybu ALIGN_4B.

Rozszerzony zestaw instrukcji dla trybu ALIGN_4B spowodował konieczność nadania im nowych identyfikatorów `it` znajdujących się w grupie `ig=1C` (por. pkt. 3.2). Negatywnym efektem nowego rozkładu danych 4B okazał się zauważalny wzrost rozmiaru kodu binarnego. Przykładowo, inicjacja jednego bajtu w trybie bez wyrównania zajmuje 8B (2B – kod instrukcji, 4B – adres, 1B – długość danych, 1B – dane), a w trybie z wyrównaniem 16B (każdy operand po 4B). Mimo usprawnień kompilatora dotyczących pracy z pamięcią, 8-bajtowe typy danych takie jak LWORD, LREAL i DATE_AND_TIME mogą powodować zwiększenie czasu wykonywania instrukcji w niektórych architekturach. Przykładowo, przy rdzeniu Cortex-M4 odczytuje się te typy za pomocą dwóch instrukcji w porcjach po 4B.

4.4. Podsumowanie wydajności i zajętości pamięci

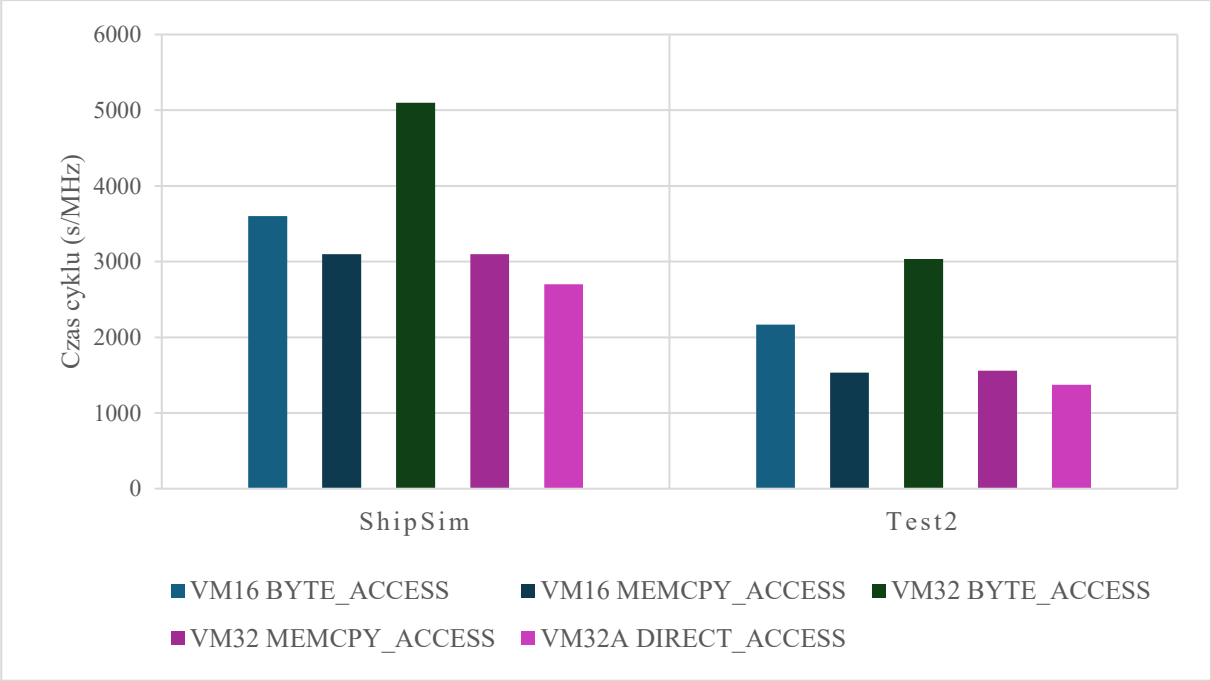
Przedstawione metody dostępu do pamięci sklasyfikowano pod względem wydajności, czyli czasu procesora potrzebnego na wykonanie pojedynczego cyklu sterowania, jak również pod względem rozmiaru kodu binarnego. Na rysunku 4.5 przedstawiono zestawienie wydajności dla trzech metod:

- BYTE_ACCESS – dostęp bajtowy,
- MEMCPY_ACCESS – kopiowanie `memcpy`,
- DIRECT_ACCESS – dostęp bezpośredni,

w różnych trybach adresacji maszyny wirtualnej:

- VM16 – adresacja 16-bitowa bez wyrównania,
- VM32 – adresacja 32-bitowa bez wyrównania,
- VM32A – adresacja 32-bitowa z wyrównaniem.

Spośród całego zbioru metoda BYTE_ACCESS okazała się najmniej wydajna. Dodatkowo, zwiększenie rozmiaru adresacji powoduje wzrost czasu cyklu średnio o 50%. Tryb MEMCPY_ACCESS polegający na decyzjach kompilatora wygenerował zbliżone wyniki niezależnie od rozmiaru adresacji (VM16 i VM32).



Rys. 4.5. Względne średnie czasy wykonywania cykli programów testowych maszyny wirtualnej 16-, 32- i 32-bitowej z wyrównaniem dla metod BYTE_ACCESS, MEMCPY_ACCESS i DIRECT_ACCESS (Cortex-M4).

Dodatkowe wyrównanie danych w metodzie ALIGN_4B oraz zastosowanie bezpośredniego dostępu DIRECT_ACCESS pozwoliło zmniejszyć ten czas do około 85% czasu wykonywania metody MEMCPY_ACCESS (obydwie metody bazują na wskaźnikach). W 32-bitowej adresacji BYTE_ACCESS charakteryzuje się prawie 2-krotnie większym czasem cyklu niż DIRECT_ACCESS. Obszerniejsze zestawienie wyników czterech wspomnianych wcześniej testów przedstawiono w [34].

Drugą kwestią oprócz wydajności czasowej jest rozmiar kodu binarnego, na który wpływ ma adresacja instrukcji oraz potrzeba wyrównywania operandów i kodu w pamięci. Rozmiary kodu cm dla rozpatrywanych maszyn wirtualnych oraz programów testowych podano w tabeli 4.3. Jak widać, w przypadku programu ShipSim rozmiar ten wyniósł 9 829 bajtów dla maszyny 16-bitowej, 15 173 w trybie niewyrównanym 32-bit i 19 108 w trybie z wyrównaniem. Kod 32-bitowy (VM32) jest więc około 50% większy od swego 16-bitowego (VM16) odpowiednika, natomiast nadmiarowe wyrównanie (VM32A) pociąga za sobą prawie dwukrotny wzrost rozmiaru kodu binarnego względem jego 16-bitowej wersji.

Oznacza to konieczność zarezerwowania wyraźnie większego obszaru pamięci dla pracy maszyny wirtualnej. Podobne proporcje odnoszą się do programu testowego Test2.

Tab. 4.3. Rozmiary pamięci kodu cm i danych dm w bajtach dla programów testowych.

Maszyna wirtualna	**ShipSim**		**Test2**	
	$\boldsymbol{cm}$	$\boldsymbol{dm}$	$\boldsymbol{cm}$	$\boldsymbol{dm}$
VM16	9 829	4 923	581	221
VM32	15 173	4 923	879	221
VM32A	19 108	4 946	1 164	224

W tabeli 4.3 podano również rozmiary pamięci danych dm dla programów testowych. Obszar ten obejmuje wartości zdefiniowanych zmiennych globalnych oraz struktury związane z blokami podstawowymi. Jak należało oczekiwać rozmiary pamięci dm dla maszyn VM16, VM32 różniących się tylko adresacją są identyczne. W odniesieniu do maszyny VM32A rozmiar dm marginalnie wzrósł wobec wyrównania wartości nielicznych zmiennych globalnych typów BOOL, INT (TEST2 zawiera tylko jedną zmienną typu BOOL).

Należy ponownie podkreślić, że wprowadzenie zmodyfikowanych instrukcji dla adresacji z wyrównaniem odbiło się negatywnie na prekompilowanych bibliotekach środowiska CPDev. Nowe procedury systemowe, poczynając od inicjacji pamięci, wymusiły wprowadzenie nowego formatu tych bibliotek, który musiał jednocześnie uwzględniać obie postacie pracujące na danych niewyrównanych i wyrównanych. Rozwiązanie to znane jest choćby z formatu Mach-O stosowanego przez firmę Apple [66], gdzie linker na podstawie architektury wybiera właściwą dołączaną przez niego postać biblioteki.

W praktyce przemysłowej nadal stosowane są układy mikroprocesorowe o różnych stopniach zaawansowania. Stąd ogólne wnioski dotyczące wyboru metod dostępu do pamięci można streścić następująco. W przypadku rozwiązań ekonomicznych lub układów starszych generacji zaleca się stosowanie metody BYTE_ACCESS dla 16-bitowej maszyny, a MEMCPY_ACCESS dla 32-bitowej. Wyrównywanie danych ALIGN_4B wraz z dostępem DIRECT_ACCESS przynosi istotny wzrost wydajności i odpowiada możliwościom nowoczesnych jednostek, jednak kosztem zwiększenia rozmiaru kodu binarnego. Ponadto, wymusza to zmiany w pozostałych modułach CPDev, w tym kompilatora i bibliotek, a więc wygenerowania nowego kodu binarnego.

Trzecim kryterium oceny, poza wydajnością i zajętością pamięci, jest efektywność energetyczna. Wyniki badań mikrokontrolerów z czterech serii STM32 przeprowadzonych według wytycznych z [67] podano w [48]. Okazuje się, że zużycie energii przez maszyny wirtualne wyrażone w $\mu a/cykl$ może się nawet parokrotnie różnić.

5. Ogólna koncepcja dwurdzeniowego sterownika programowalnego

W rozdziale przedstawiono koncepcję rozszerzenia typowego środowiska programistyczno-wykonawczego IDE, odpowiedniego dla jednozadaniowego projektu klasy PLC, na sterownik dwurdzeniowy wykonujący na swych rdzeniach dwa różne projekty w trybie bare-metal, tzn. bez pośrednictwa wielozadaniowego systemu operacyjnego czasu rzeczywistego RTOS. W zamian sterownik dwurdzeniowy dysponuje pamięcią współdzieloną do wymiany danych między rdzeniami. Sterownik taki, w którym projekty są wykonywane przez rdzenie jako odpowiedniki zadań, mógłby zastąpić sterownik z systemem RTOS. Zgodnie z normą IEC 61131-3 współpraca między zadaniami polega na wymianie zmiennych globalnych. Na przykładzie sterownika Freelance ABB [5] pokazano jak dostęp do tych zmiennych wygląda w rozproszonym systemie sterowania DCS. Drugim przedstawionym przykładem jest sterownik z serii CX Beckhoffa [4] klasyfikowany jako Embedded PC. Okazuje się, że kluczowym warunkiem dla części programistycznej środowiska IDE rozszerzanego na dwa rdzenie jest jednakowa lista wymienianych zmiennych globalnych deklarowana w projektach dla tych rdzeni. Ponadto zmienne te powinny w każdym projekcie otrzymać dodatkowy atrybut wskazujący czy mają być odczytywane, czy zapisywane w pamięci współdzielonej. Natomiast w części wykonawczej, oprócz obsługi pamięci współdzielonej, istotną sprawą jest unikanie konfliktów powstających, gdyby obywa rdzenie jednocześnie żądały do niej dostępu. Przedstawioną koncepcję można uogólnić na sterownik o większej liczbie rdzeni.

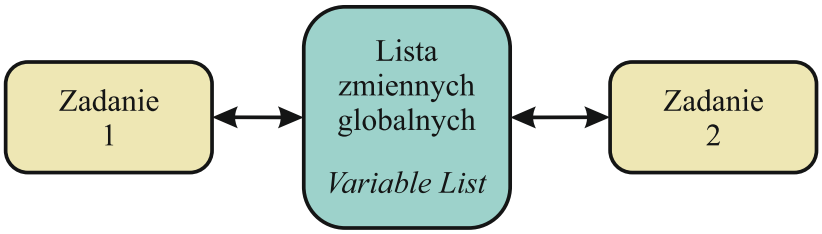
5.1. Zmienne globalne w sterownikach z systemami RTOS

RTOS i DCS są kluczowymi rozwiązaniami stosowanymi w rozbudowanych aplikacjach wymagających kontroli procesów. RTOS jako system operacyjny czasu rzeczywistego zarządza procesami i zasobami w sposób deterministyczny, zapewniając wykonywanie zadań w określonych ramach czasowych [68]. Przykładami firm produkujących sterowniki z systemami RTOS są Beckhoff, Siemens, Schneider czy WAGO. Z kolei DCS oznacza system sterowania kontrolujący rozbudowane procesy przemysłowe poprzez sieć rozproszonych sterowników również wyposażonych w RTOS. Czołowymi producentami systemów DCS są ABB, Siemens, Honeywell i Alstom.

Pierwsze z dwóch przedstawionych poniżej przykładowych rozwiązań dotyczy sterowników serii Freelance z koncernu ABB, do której obecnie należą AC 700/800/900 F [69],

poprzednio Freelance 2000, a jeszcze wcześniej Digimatik wprowadzony przez zasłużoną dla automatyki i pomiarów firmę Hartmann-Braun (wtedy niezależną). Sterowniki Freelance są przeznaczone dla systemów sterowania DCS średniej wielkości, które w Polsce znajdują się w zakładach azotowych, elektrociepłowniach, cukrowniach i in. Szeroka akceptacja przez przemysł w wielu krajach sprawiła, że koncepcja oprogramowania Freelance pozostała w znacznym stopniu taka sama jak w oryginalnym Digimatiku.

Środowisko inżynierskie Freelance Engineering (poprzednio Control Builder F, a wcześniej DigiTool) bazuje na dedykowanym systemie operacyjnym RTOS z 16 zadaniami, w tym 9 zadań dla użytkownika (8 cyklicznych, 1 domyślne – *default*) oraz 7 zadań systemowych [5]. W programach POU składających się na zadanie, deklarowane są zmienne lokalne VAR oraz zmienne zewnętrzne VAR_EXTERNAL. Źródłem zmiennych VAR_EXTERNAL mogą być inne programy tego samego zadania, programy z innych zadań, bądź moduły I/O i komunikacyjne. Istotne jest, że deklarowane zmienne VAR_EXTERNAL automatycznie włączane są do listy zmiennych globalnych *Variable List*, skąd są dostępne we wszystkich zadaniach sterownika, a także w pozostałych stacjach systemu DCS. Zatem VAR_EXTERNAL może tu być utożsamiane z VAR_GLOBAL w sensie normy IEC 61131-3. Ogólna organizacja przetwarzania dla dwóch zadań wygląda jak na rysunku 5.1.



Rys. 5.1. Zadania i zmienne globalne VAR_EXTERNAL w *Variable List* – Freelance.

Domyślnym trybem dostępu do zmiennych globalnych w *Variable List* jest obraz procesu (*process image*), sygnalizowany już w punkcie 3.5. Tak więc, na początku wykonywania zadania system operacyjny kopiuje z *Variable List* wartości zmiennych globalnych do roboczego obszaru *process image*. Następnie dokonuje obliczeń i w tym obszarze składa wyniki. Na końcu zadania system aktualizuje *Variable List* według zawartości *process image*. Rozmiar *process image* jest jednym z parametrów zadania (domyślnie 8 kB).

Drugie z przedstawionych rozwiązań dotyczy sterowników PLC firmy Beckhoff Automation. Po sterownikach Siemensa są one jednymi z najczęściej spotykanych w Polsce. Aktualnie wiodące sterowniki tej firmy należą do serii CX20/50/52/70/90XX [70]. Beckhoff traktuje je jako komputery klasy Embedded PC, ponieważ umożliwiają m.in. tworzenie rozproszonych systemów kontrolno-pomiarowych z „wyspami I/O” połączonymi szybką

magistralą EtherCAT. W zależności od serii, wyposażone są one w procesory ARM lub x86 (Atom, i3 i in.) z systemami operacyjnymi Windows Embedded Compact 7, Windows 10 IoT Enterprise, TwinCAT/RTOS lub TwinCAT/BSD.

Środowisko inżynierskie TwinCAT 3 [4], częściowo oparte jest na szeroko stosowanym środowisku CODESYS [71]. Oprogramowanie to nie ma sztywnego limitu liczby zadań, które można utworzyć w systemie. Istnieją jednak praktyczne ograniczenia wynikające głównie z dostępnych zasobów sprzętowych. Poza tym, każde zadanie charakteryzuje się czasem cyklu (*task cycle time*), będącego wielokrotnością czasu bazowego (*base time*) oraz priorytetem w zakresie od 1 do 61. Jedną z wyróżniających się cech jest możliwość przydzielania zadań wykonywanych na wybranym, izolowanym lub współdzielonym z systemem operacyjnym wątku procesora.

Zasięg zmiennych lokalnych deklarowanych za pomocą słowa kluczowego VAR w obrębie jednostek programowych POU jest ograniczony do pojedynczej POU. Zmienne do komunikacji międzyprogramowej lub międzyzadaniowej w środowisku TwinCAT 3 umieszczane są w specjalnych plikach GVL (*Global Variable List*) zawierających listy zmiennych globalnych. W związku z tym, wykonywanie obliczeń w sterownikach Beckhoffa można przedstawić jak na rysunku 5.2 (w sumie zbliżonym do rys. 5.1). W odróżnieniu jednak od środowiska Freelance, gdzie zmienne globalne są utożsamiane ze wszystkimi VAR_EXTERNAL deklarowanymi w systemie DCS, tutaj listy w plikach GVL obejmują zestawy zmiennych VAR_GLOBAL, które mogą być używane w programach i zadaniach. Zapewnia to więc większą elastyczność i spójność.

Rys. 5.2. Zadania i zmienne globalne GVL – TwinCAT.

W systemie wielozadaniowym współdzielenie zmiennych globalnych wiąże się z ryzykiem związanym ze współbieżnym dostępem. Niektóre systemy, jak TwinCAT, nie zabezpieczają automatycznie dostępu do wspólnych zasobów, pozostawiając to programiście. Do realizacji zabezpieczenia udostępniane są mechanizmy synchronizacji, takie jak muteks, funkcje atomowe i priorytety zadań, scharakteryzowane krótko w punkcie 5.4. Dzięki tym mechanizmom możliwa jest odpowiednia synchronizacja zmiennych globalnych,

zapobiegająca ryzyku hazardu (*race condition*) dotyczącego niedeterministycznego dostępu do wspólnych zasobów. Problem ten występuje, gdy kolejność wykonywania operacji nie jest przewidywalna, co może prowadzić do konfliktu przy jednoczesnym zapisie i odczycie tych samych obszarów pamięci.

5.2. Podstawowe warunki dla projektów dwurdzeniowych

Procesor wielordzeniowy jest układem scalonym, składającym się z rdzeni, będących jednostkami CPU, mogącymi niezależnie przetwarzać dane. Wzrasta dzięki temu ogólna wydajność takiego procesora. Współpracę rdzeni koordynuje program zarządzający, a do komunikacji międzyrdzeniowej służy pamięć współdzielona bądź technologie magistralowe. Dostęp do pamięci współdzielonej może być synchronizowany za pomocą semaforów. Dwu- lub czterordzeniowe procesory o architekturze ARM dominują w urządzeniach wbudowanych i mobilnych, natomiast wielordzeniowe architektury x86 są przeznaczone dla komputerów PC i serwerów.

Wielordzeniowość w architekturze ARM dotyczy zarówno mikroprocesorów (seria Cortex-A) jak i mikrokontrolerów (Cortex-M) scharakteryzowanych w punkcie 4.1. W ostatnich latach szczególne zainteresowanie budzą układy hybrydowe, w których w ramach jednego układu SoC (*System on Chip*) łączone są mikroprocesor, przeznaczony do obsługi systemu operacyjnego i jego aplikacji, oraz mikrokontroler, zapewniający deterministyczne sterowanie czasowe. Stąd wielordzeniowość może mieć charakter homogeniczny lub heterogeniczny – hybrydowy. Przykłady układów wielordzeniowych podano w tabeli 5.1.

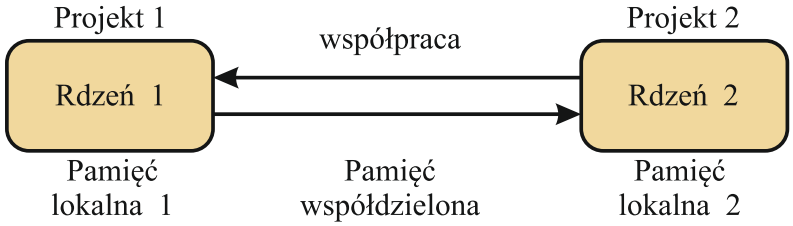
Tab. 5.1. Przykłady wielordzeniowych układów scalonych.

Typ	**Model**
Mikrokontroler	STMicroelectronics STM32H755 (1x Cortex-M4 + 1x Cortex-M7) Raspberry Pi Foundation RP2040 (2x Cortex-M0+)
Mikroprocesor	Texas Instruments AM5729 (2x Cortex-A15) Broadcom BCM2712 (4x Cortex-A76)
Heterogeniczny	STMicroelectronics STM32MP1 (1x Cortex-M4 + 1/2x Cortex-A7) NXP Semiconductors i.MX.8M (1x Cortex-M4 + 4x Cortex-A53)

Istotne różnice między platformami wielordzeniowymi doprowadziły do opracowania uniwersalnego środowiska OpenAMP (*framework*) [72], które udostępnia komponenty oprogramowania standaryzujące interakcje między rdzeniami. Dotyczy to jednak rdzeni mikroprocesorowych korzystających z systemów operacyjnych, jak Linux czy FreeRTOS. OpenAMP nie jest jednak potrzebny w przypadku programowania mikrokontrolerów w trybie *bare-metal*, w którym programista odpowiada za synchronizację zadań, ochronę zasobów i inne

mechanizmy, korzystając tylko z narzędzi i bibliotek dostarczonych przez producenta układu scalonego. Jak podano we Wprowadzeniu (rozdz. 1), tryb *bare-metal* jest typowy dla niewielkich urządzeń automatyki, w których prostota i niskie koszty są kluczowe. Z tych powodów założono, że w prezentowanych w pracy rozwiązaniach tryb *bare-metal* jest bazą dla implementacji projektów IEC 61131-3.

Przedstawioną poniżej koncepcję dwurdzeniowego sterownika programowalnego, oryginalnie zaproponowaną w [35], można traktować jako odpowiadającą dwóm sterownikom połączonym pewnym łączem komunikacyjnym. W sterowniku dwurdzeniowym jako łącze służy pamięć współdzielona. Ogólną architekturę takiego sterownika pokazano na rysunku 5.3, gdzie projekty 1 i 2 reprezentują kod wykonywany przez rdzenie. Zakłada się, że każdy z rdzeni może pracować niezależnie, korzystając z własnej pamięci lokalnej.



Rys. 5.3. Ogólna architektura dwurdzeniowego sterownika programowalnego.

Projekty wykonywane przez rdzenie współpracują poprzez wymianę zmiennych globalnych, które można oznaczyć następująco:

$GV1_R$ – zmienne odczytywane (READ) w projekcie 1 i zapisywane (WRITE) przez projekt 2,

$GV1_W$ – zmienne zapisywane w projekcie 1 i odczytywane w projekcie 2.

Podobnie w projekcie 2 mamy $GV2_R$ i $GV2_W$. Zakłada się, że zmienne $GV1_R$ odczytywane w projekcie 1 są takie same jak zapisywane $GV2_W$ w projekcie 2. Podobnie $GV2_R$ odpowiada $GV1_W$. Stąd zmienne globalne GV wymieniane między projektami można zapisać jako zbiór

$$GV = GV1_R \cup GV1_W = GV2_R \cup GV2_W \tag{5.1}$$

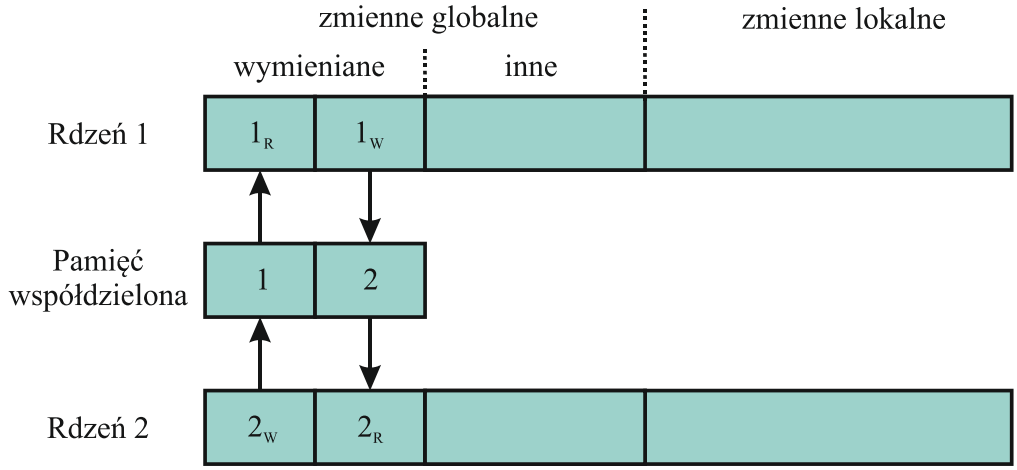
Zatem współpraca projektów sterowania zgodnych z normą IEC 61131-3 wymaga spełnienia następującego, podstawowego warunku:

Deklaracja zmiennych globalnych w każdym z dwóch współpracujących projektów musi zawierać wszystkie wymieniane zmienne, czyli zbiór GV.

Ponieważ rozpatrywane typowe środowisko IDE (niekoniecznie CPDev) jest z założenia przeznaczone dla sterownika jednordzeniowego, spełnienie powyższego warunku wymaga wskazania, które zmienne są odczytywane w danym projekcie, a które zapisywane.

Potrzebne są dodatkowe atrybuty określające tryb dostępu do wymienianych zmiennych globalnych, czyli tutaj READ i WRITE.

Jest to drugi warunek rozszerzenia środowiska IDE. Na tej podstawie strukturę pamięci sterownika dwurdzeniowego można przedstawić jak na rysunku 5.4, gdzie 1_R, 1_w oraz 2_R, 2_w oznaczają sektory pamięci odpowiednich rdzeni, a 1 i 2 (bez indeksu) sektory w pamięci współdzielonej. Zgodnie z rysunkiem, odpowiednia kolejność i podział pamięci na jednolite sektory są kluczowe dla płynnej synchronizacji pamięci lokalnej i współdzielonej.



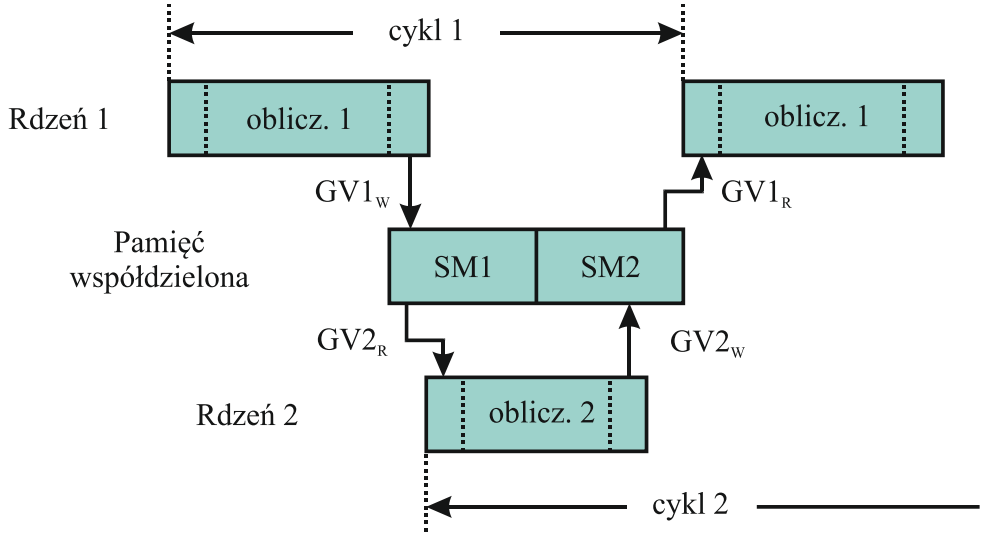
Rys. 5.4. Struktura pamięci sterownika dwurdzeniowego.

5.3. Organizacja pracy sterownika dwurdzeniowego

Sterowniki programowalne z zasady pracują w trybie cyklicznym. Jak wskazano wcześniej (pkt. 3.5), podczas wykonywania cyklu obliczeń stosowany jest tryb odczyt-wykonywanie-zapis w odniesieniu do zmiennych wejściowych i wyjściowych (jak *process image* powyżej). Według nazewnictwa zastosowanego w punkcie 3.5, aktualizacja zmiennych wejściowych wykonywana jest na początku cyklu w fazie *precycle*, a wyjściowych na końcu w *postcycle*. Warto jeszcze dodać, że możliwy jest również tryb asynchroniczny, w którym wykonywanie obliczeń i operacji sterujących jest inicjowane przez zdarzenia, takie jak żądanie z zewnętrznego systemu, zmiana stanu wejść, bądź wystąpienie przerwania sprzętowego. W nowych sterownikach PLC łączy się często oba te tryby, tzn. synchronizacja głównej pętli sterującej jest uzupełniana asynchroniczną obsługą przerwań i zdarzeń.

Organizację współpracy za pośrednictwem pamięci współdzielonej dwóch cyklicznie wykonywanych projektów, umieszczonych w osobnych rdzeniach, pokazano na rysunku 5.5, oddzielając linią przerywaną fazy *precycle* i *postcycle* w każdym z cykli. Jak widać, rdzeń 1 po zakończeniu obliczeń, w swym *postcycle* wykonuje operację $GV1_W \rightarrow SM1$, zapisując zaktualizowane zmienne do sektora $SM1$ w pamięci współdzielonej (SM – *shared memory*). Zawartość $SM1$ jest odczytywana przez rdzeń 2 w swym *precycle* operacją $SM1 \rightarrow GV2_R$, ale dopiero wtedy, gdy program zarządzający odpowiednio do cyklu 2 aktywuje ten rdzeń. Odczyt $GV2_R$ jest więc zwykle opóźniony w stosunku do zapisu $GV1_w$, co sygnalizuje przełamana strzałka na rysunku 5.5. Analogicznie odczyt $GV1_R$ w rdzeniu 1 jest opóźniony względem zapisu $GV2_w$ przez rdzeń 2.

Organizacja współpracy rdzeni poprzez fazy precycle i postcycle w obydwu cyklach jest trzecim warunkiem dla pracy sterownika dwurdzeniowego.



Rys. 5.5. Współpraca rdzeni z uwzględnieniem faz *precycle* i *postcycle*.

Skrócenie czasu synchronizacji zmiennych jest istotne dla działania sterownika, przy czym podział pamięci na jednolite sektory wspomaga płynne kopiowanie. Dodatkowym sposobem przyśpieszenia kopiowania może być wykorzystanie mechanizmu transferu danych DMA (*Direct Memory Access*), w tym przypadku w trybie zwanym pamięć do pamięci (*mem-to-mem*). Pozwala to na kopiowanie danych bez bezpośredniego udziału procesora, jednak wymaga pewnego czasu na zainicjowanie kontrolera DMA. Z tego powodu mechanizm DMA jest preferowany przy kopiowaniu większych obszarów pamięci, natomiast w przypadku małych fragmentów może okazać się on nieefektywny.

Celem uniknięcia konfliktów w dostępie do pamięci współdzielonej, mikrokontrolery wielordzeniowe wyposażone są w sprzętowe semafory (flagi), pozwalające sprawdzić, czy dostęp jest aktualnie możliwy. Przykładami są semafory HSEM w STM32 [36] oraz mechanizm Spinlock w RP2040/RP2350 [28]. Program wykonawczy (*runtime*) musi więc zawierać algorytm obsługi takiego semafora przez każdy rdzeń.

Dostęp do pamięci współdzielonej należy uzupełnić o algorytm unikania konfliktów przy zapisie i odczycie.

Jest to czwarty warunek rozszerzenia środowiska IDE na dwa rdzenie. Odpowiedni algorytm i kod w odniesieniu do mikrokontrolera STM32 został podany w następnym rozdziale.

W sumie, na ogólną koncepcję rozszerzenia typowego jednozadaniowego środowiska IDE na wykonywanie dwóch projektów IEC 61131-3 przez procesor dwurdzeniowy składają się:

- w części programistycznej:
  - takie same listy zmiennych globalnych wymienianych między projektami,
  - zamienne atrybuty READ/WRITE tych zmiennych w obydwu projektach,
- w części wykonawczej:
  - wymiana zmiennych przez pamięć współdzieloną w fazach *precycle* i *postcycle* cykli obydwu rdzeni,
  - eliminacja konfliktów w dostępie rdzeni do pamięci współdzielonej.

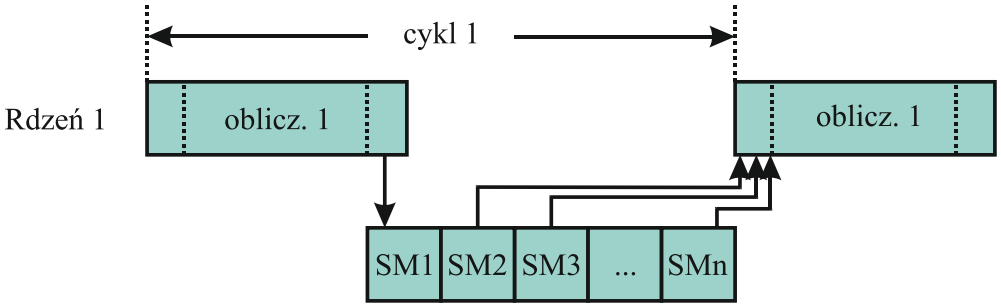
5.4. Uogólnienie dla procesora wielordzeniowego

Przedstawioną powyżej koncepcję sterownika dwurdzeniowego można uogólnić na procesor wielordzeniowy, w którym rdzenie przeznaczone do sterowania wykonują osobne projekty (zadania). W praktyce więcej niż dwa rdzenie zawierają przede wszystkim układy mikroprocesorowe, których oprogramowanie zazwyczaj jest uruchamiane w systemie operacyjnym (Linux lub FreeRTOS).

W przypadku czterech rdzeni, jak w układach mikrokomputerów Raspberry Pi, rdzeń 1 mógłby np. wykonywać sterowanie logiczne (zwykle co 10 ms), rdzeń 2 prowadzić regulację PID (100 ms), rdzeń 3 kontrolować ograniczenia procesowe (1 s), a rdzeń 4 za pomocą systemu Linux obsługiwać bazę danych i komunikację (rozdz. 7). Odpowiadałoby to typowym zadaniom sterowników w niewielkich systemach SCADA i DCS.

Niezależnie od typu układu i struktury oprogramowania, w przedstawianym rozwiązaniu listy wymienianych zmiennych globalnych powinny być jednakowe we

wszystkich projektach, a pamięć współdzielona składać się z sektorów SM1, SM2 itd., do których kolejne rdzenie zapisują zmienne odczytywane przez pozostałe rdzenie. Tak więc, jak pokazano na rysunku 5.6, rdzeń 1 w swym *postcycle* zapisuje swój sektor SM1, a w *precycle* odczytuje zmienne z zapisanych wcześniej przez inne rdzenie sektorów SM2, SM3 itd. Przełamane strzałki wskazują na opóźnienia między zapisem a odczytem. Należy dodać, że zgodnie z wytycznymi programowania dla IEC 61131-3 [73] atrybut WRITE wymienianej zmiennej globalnej powinien być unikalny, tzn. można go jej przyporządkować tylko raz.



Rys. 5.6. *Precycle* i *postcycle* dla rdzenia 1 w procesorze wielordzeniowym.

Pierwowzorem dla przedstawionej koncepcji sterownika wielordzeniowego była realizacja w układzie FPGA [74], z maszynami wirtualnymi CPDev w każdym ze skonfigurowanych rdzeni. Porównywano z nią czasy wykonywania programów testowych (logika, PID, sterowanie rozmyte) z czasami dla sterowników PLC (GE Fanuc, Siemens, Beckhoff). Wielordzeniowe implementacje FPGA, ale z IL jako językiem pośrednim, są opisane w [17,75]. Procesor dwurdzeniowy może także służyć do edukacji wspieranej symulacjami. Na przykład, jeden z rdzeni rozwiązania prezentowanego w [76] funkcjonuje jako symulator obiektu (zbiornik, przetwornica), a drugi jako sterownik. Programy są tam opracowywane w C. Można jeszcze dodać, że porównanie czasu wykonywania dwóch programów IEC 61131-3 przez sterownik z jednym lub dwoma rdzeniami podano w [77] (bez wymiany zmiennych między rdzeniami).

W sterowniku wielordzeniowym nierównomierny dostęp do pamięci współdzielonej może uniemożliwić realizację zadań w czasie rzeczywistym. Ze względu na ryzyko niezgodności danych wynikające z jednoczesnych prób odczytu i zapisu tego samego obszaru pamięci, w systemach RTOS, takich jak TwinCAT 3, synchronizacja zmiennych globalnych może być realizowana za pomocą następujących mechanizmów [78,4]:

- muteks (*mutual exclusion*), który pozwala na zabezpieczenie sekcji pamięci, zapewniając tylko jednemu zadaniu w danym momencie dostęp do określonych zmiennych globalnych,

- funkcje atomowe, takie jak `TestAndSet()`, pozwalają na bezpieczną, nienaruszalną zmianę stanu zmiennej,
- priorytety zadań – TwinCAT i Freelance pozwalają na przypisanie zadaniom priorytetów, co może minimalizować ryzyko konfliktów przy dostępie do zmiennych.

Mechanizm muteks systemu Linux zastosowano w rozdziale 7 do eliminacji konfliktów w dostępie do pamięci współdzielonej rdzeni mikrokomputera Raspberry Pi.

Podsumowując, realizację zadań czasu rzeczywistego można przeprowadzić za pomocą systemu operacyjnego lub poprzez przydzielenie zadań do konkretnych rdzeni procesora. Rozwiązanie wielordzeniowe może w prostych przypadkach stanowić alternatywę dla systemów opartych na systemach RTOS. Wadą jest jednak ograniczenie liczby zadań związane z dostępną liczbą rdzeni.

6. Rozszerzenie i implementacja środowiska CPDev dla dwóch rdzeni

W rozdziale przedstawiono rozwiązania rozszerzające środowisko CPDev o wykonywanie dwóch projektów IEC 61131-3 przez dwa rdzenie procesora. Projekty mogą być tworzone osobno, pod warunkiem jednakowych deklaracji wymienianych zmiennych globalnych i zamiennych atrybutów READ/WRITE w obydwu projektach. Alternatywą jest utworzenie najpierw wspólnego projektu z deklaracją wszystkich zmiennych globalnych, zarówno wymienianych jak i pozostałych, wraz ze wszystkimi programami POU przewidzianymi do wykonywania. Na etapie przydzielania zadań do rdzeni następuje wybór programów do jednego lub drugiego zadania, po którym następuje kompilacja. Ze względu na różnicę cykli wykonywania projektów, do symulacji pracy sterownika dwurdzeniowego wykorzystano wielowątkowe środowisko WinController [37] oraz symulatory CPSim (pkt. 3.4, [19]). Obsługa pamięci współdzielonej w fizycznym sterowniku wymaga uzupełnienia języka pośredniego VMASM o procedury zapisu i odczytu, z zabezpieczeniem przeciw kolizji. Podano denotacyjny model jednej z tych procedur uwzględniający zabezpieczenie. Do praktycznej implementacji wykorzystano dwurdzeniowy mikrokontroler STM32 w płycie uruchomieniowej Nucleo [36]. Przedstawiono strukturę kodu C/C++ pętli rdzenia z dostępem do pamięci współdzielonej za pomocą semafora, a także wybrane szczegóły techniczne.

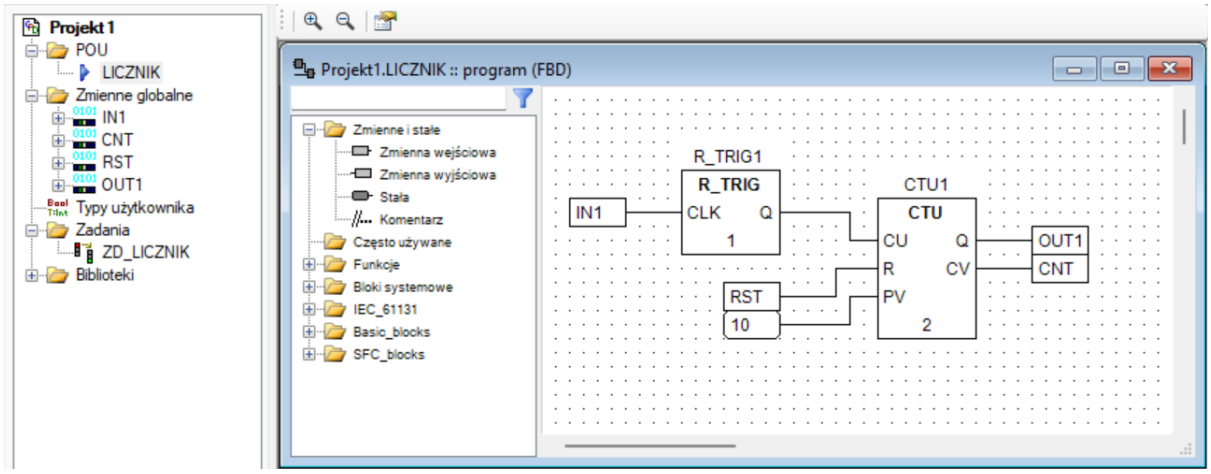
6.1. Osobne projekty dla rdzeni

Rozszerzenie części IDE środowiska CPDev zademonstrowano najpierw na przykładzie dwóch osobnych projektów pokazanych na rysunkach 6.1a,b, pierwszym utworzonym w języku graficznym FBD, a drugim w tekstowym ST (jak w [35]). Wspólnym celem projektów jest zliczanie impulsów, monitorowanie czasu ich trwania i przekroczeń na podstawie wymienianych zmiennych globalnych. W szczególności w programie LICZNIK Projektu 1 (rys. 6.1a) bramkowy impuls IN1 z wejścia I/O lub symulatora CPSim jest podawany na detektor narastającego zbocza R_TRIG1, a stąd na inkrementacyjny licznik CTU1, z liczbą impulsów wskazywaną na wyjściu CNT (R_TRIG1 i CTU1 są instancjami standardowych bloków IEC 61131-3). Drugie wyjście OUT1 licznika wskazuje czy CNT osiągnęło granicę 10. Licznik może być ponadto zerowany sygnałem RST pochodzącym z Projektu 2.

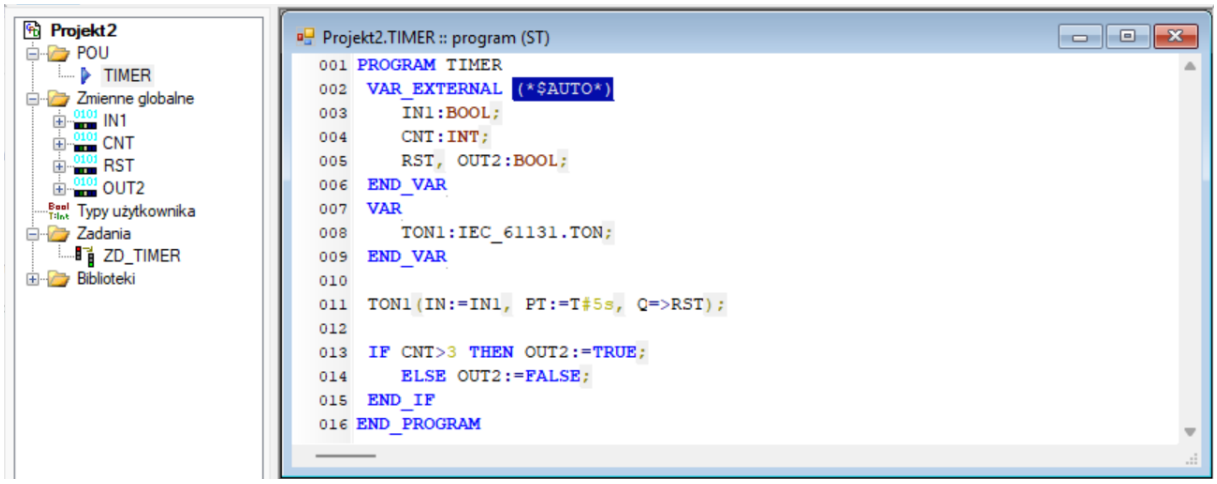
Program TIMER w Projekcie 2 (rys. 6.1b) zawiera czasomierz TON1 mierzący czas trwania impulsu IN1 otrzymanego z Projektu 1. Jeżeli IN1 trwa przynajmniej 5 sekund, to wyjście Q czasomierza przechodzi w stan wysoki i poprzez RST zeruje licznik CTU1

w Projekcie 1. Niezależnie od tego, Projekt 2 poprzez OUT2 sygnalizuje czy CNT przekroczyło próg 3.

a)



b)



Rys. 6.1. Osobne projekty dla rdzeni: a) Projekt 1 w FBD, b) Projekt 2 w ST.

Zmiennymi wymienianymi w projektach są IN1, CNT i RST. Atrybuty READ/WRITE powinny być takie jak pokazano w tabeli 6.1.

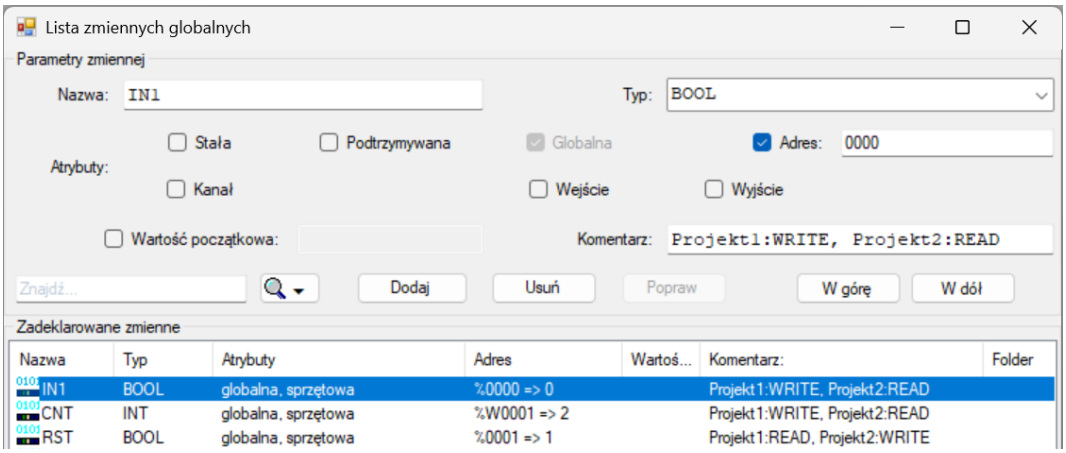
Tab. 6.1. Atrybuty READ, WRITE wymienianych zmiennych globalnych.

Zmienna	**Projekt1**	**Projekt2**
IN1	WRITE	READ
CNT	WRITE	READ
RST	READ	WRITE

Atrybuty te definiuje się w oknie *Lista zmiennych globalnych* środowiska IDE pokazanej na rysunku 6.2 w polu *Komentarz* odpowiednio do tej tabeli. Brak komentarza oznacza, że dana zmienna globalna nie jest zmienną wymienianą. Atrybuty READ/WRITE poprzedzone

nazwami projektów mieszczą się w pliku *.dcp* w sekcji danej zmiennej. Korzystają z nich maszyny wirtualne obsługujące pamięć współdzieloną w fazach *precycle* i *postcycle* (rys. 5.5).

Po zdefiniowaniu czasów cykli w oknach *Właściwości zadania* (zob. dalej) obydwa projekty są kompilowane jako zadania, generując pliki binarne dla dwóch rdzeni.

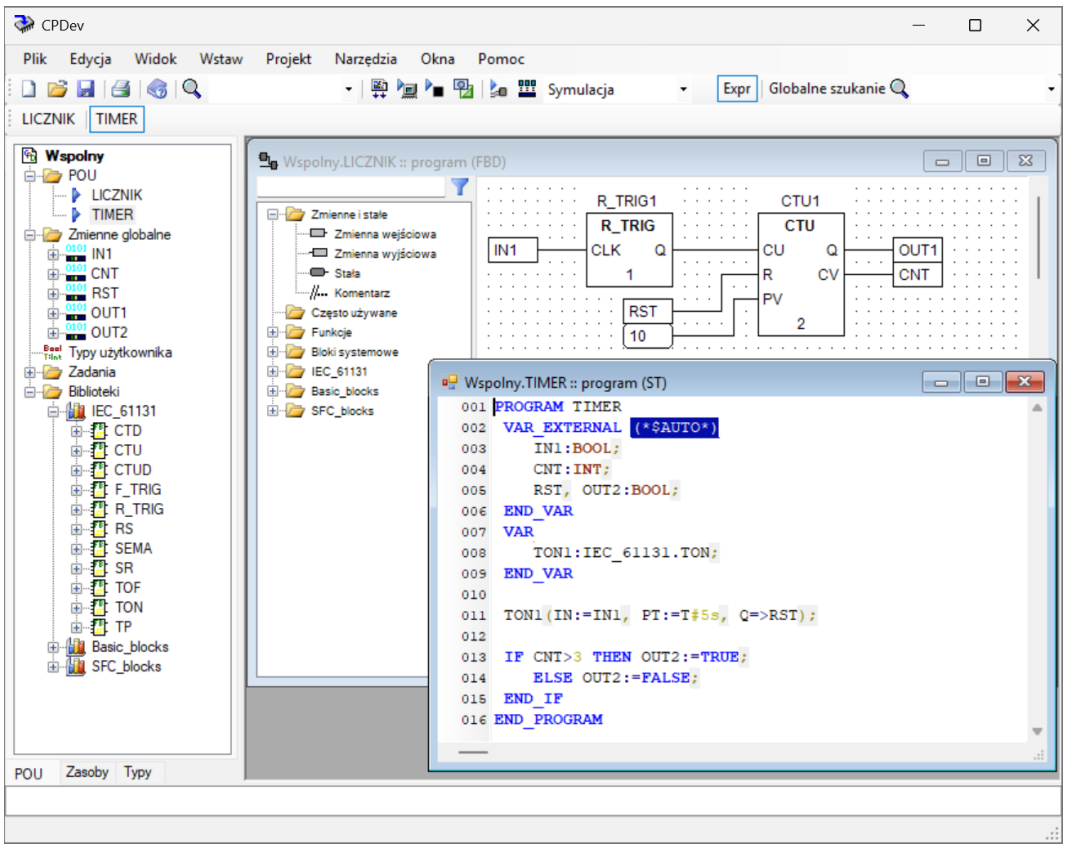


Rys. 6.2. Deklaracja wymienianych zmiennych globalnych w obydwu projektach.

6.2. Wspólny projekt dla dwóch rdzeni

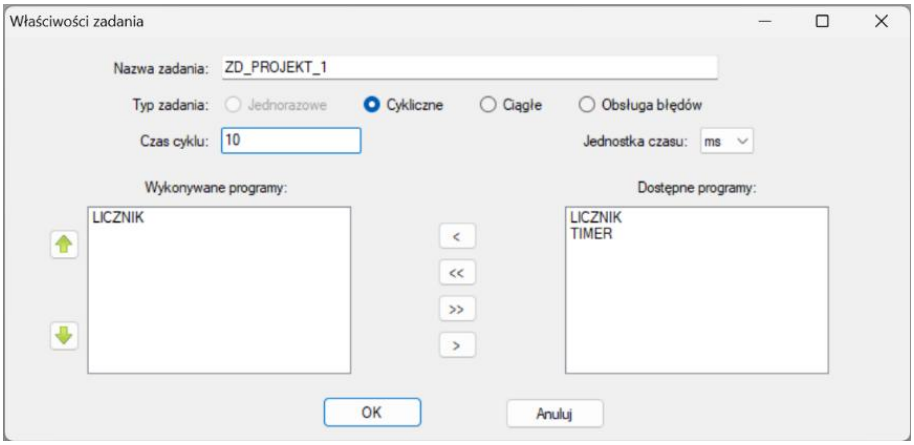
Alternatywną dla osobnych projektów może być jeden wspólny projekt, który dopiero przed kompilacją zostaje rozdzielony na dwie części w formie zadań. Rozwiązanie takie może ułatwić zrozumienie całego projektu, a ponadto być wygodniejsze w przypadku projektów ze znaczną liczbą wymienianych zmiennych. Przykładowy projekt o nazwie Wspolny, pokazany na rysunku 6.3 składa się z programów LICZNIK i TIMER takich jak poprzednio, ale zawiera jedną wspólną listę zmiennych globalnych, na której oprócz wymienianych zmiennych znajdują się również pozostałe zmienne, czyli OUT1 i OUT2. Atrybuty READ/WRITE wymienianych zmiennych wpisuje się w polach *Komentarz* w oknie *Lista zmiennych globalnych* tak samo jak poprzednio.

Rozdział projektu Wspolny na dwa zadania odbywa się za pośrednictwem okien *Właściwości zadania* pokazanych na rysunkach 6.4a,b. Najpierw tworzone jest zadanie ZD_PROJEKT_1 odpowiadające Projektowi 1, do którego spośród dostępnych programów LICZNIK i TIMER do wykonywania wybierany jest LICZNIK (rys. 6.4a). Ustawiany jest wówczas czas cyklu, np. 10 ms, a potem aktywowana kompilacja, po której wynikowy plik wykonywalny można zaimplementować w rdzeniu 1. Następnie dla zadania ZD_PROJEKT_2 wybierany jest program TIMER i ustawiany cykl 50 ms (rys. 6.4b), by po ponownej kompilacji otrzymać plik wykonywalny dla rdzenia 2.

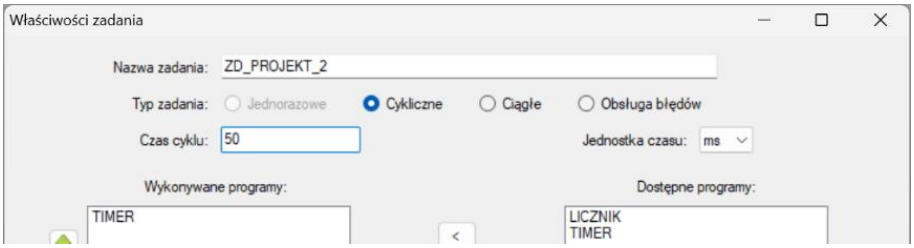


Rys. 6.3. Wspólny projekt dla obydwu rdzeni.

a)



b)

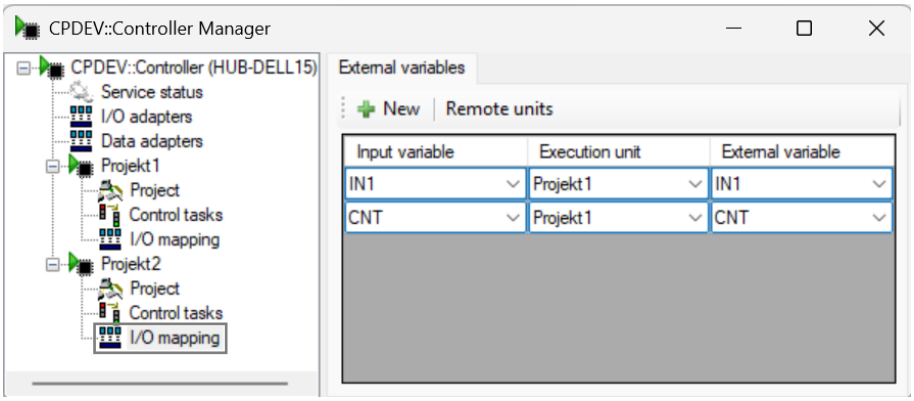


Rys. 6.4. Program wykonywany w zadaniu: a) ZD_PROJEKT_1, b) ZD_PROJEKT_2.

6.3. Symulacja współpracy rdzeni za pomocą WinControllera

CPDev::WinController, czyli WinController, jest oprogramowaniem komputera PC wykonującym zespół projektów CPDev za pomocą maszyn wirtualnych przyporządkowanych wątkom systemu Windows [37]. Komputer może pełnić wtedy rolę komputera nadrzędnego w systemie rozproszonym o łagodnych wymaganiach czasu rzeczywistego [79]. Maszyny wykonujące projekty mogą wymieniać między sobą dane podobnie jak w procesorze wielordzeniowym, dzięki czemu WinController może być platformą do symulacyjnego testowania. Maszyny te nazywane są tutaj jednostkami wykonawczymi (*Execution unit*). Pod względem technicznym WinController stanowi implementację usługi *Windows Service* [80] systemu operacyjnego.

Sposób wykorzystania WinControllera do symulacji przedstawionych projektów ilustruje rysunek 6.5. Po lewej stronie okna menadżera znajdują się dwa projekty poprzedzone statusem usługi, interfejsami *I/O adapters* kart I/O (National Instruments, Inteco) oraz adapterami *Data adapters* protokołów komunikacyjnych (Modbus TCP) dla sterowników w systemie rozproszonym. Opcja *I/O mapping* danego projektu, za pomocą której konfiguruje się wymianę zmiennych z innymi projektami, jest przystosowana do odczytywania wartości zmiennej *Input variable* tego projektu ze zmiennej *External variable* drugiego projektu wykonywanego przez inną jednostkę *Execution unit*.

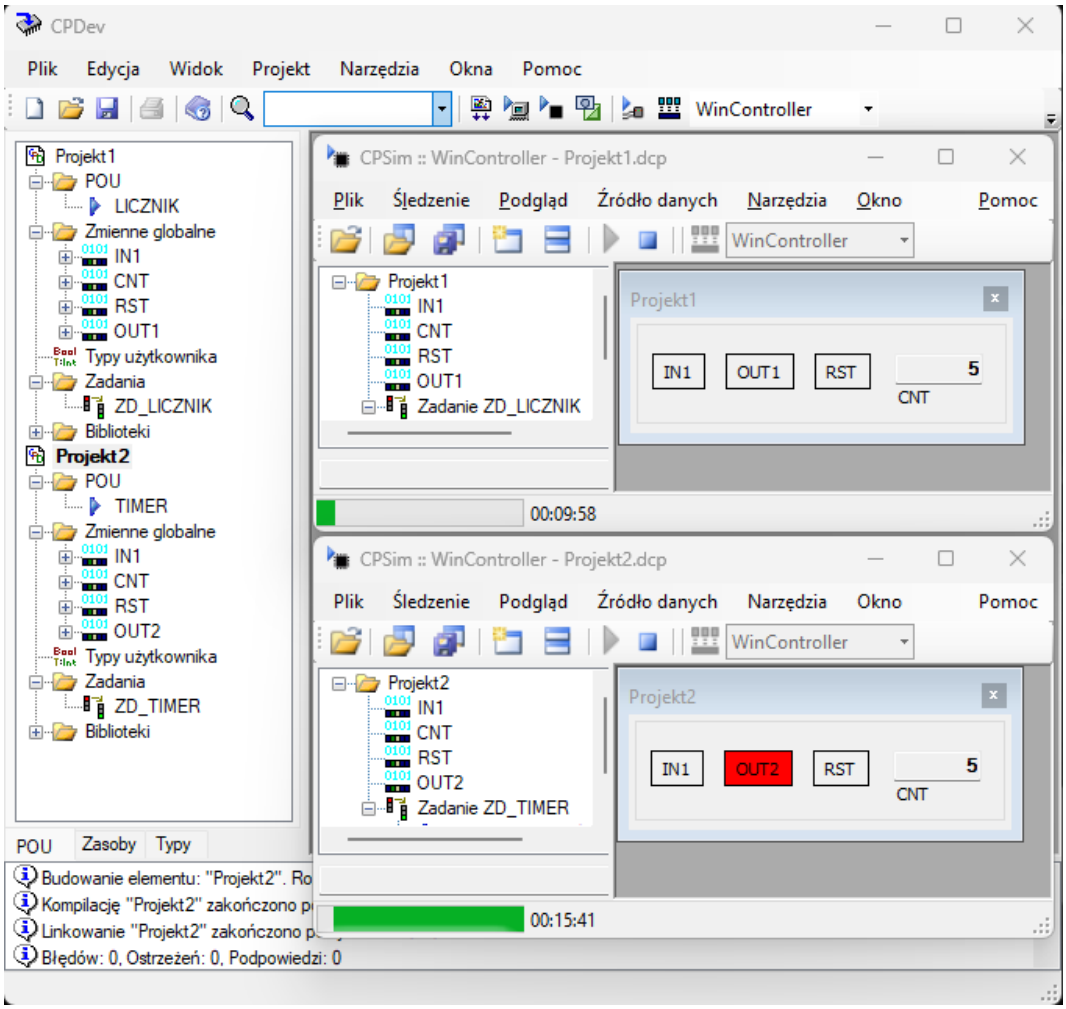


Rys. 6.5. Jednostki wykonawcze Projekt1, Projekt2 oraz wymieniane zmienne IN1, CNT jako *Input* i *External*.

Zgodnie z tym, prawa strona rysunku 6.5 przedstawia konfigurację odczytu (READ) zmiennych IN1, CNT (*Input variable*) przez Projekt 2 wybrany w *I/O mapping* (zaznaczenie), ze zmiennych również o nazwach IN1, CNT (*External variable*) zapisywanych (WRITE) w Projekcie 1 (*Execution unit*). Występowanie tej samej nazwy zmiennej zarówno jako *Input* jak i *External variable* jest charakterystyczne dla zmiennych wymienianych między projektami.

Podobnie, w przypadku odczytu zmiennej RST (*Input variable*) przez Projekt 1 (*I/O mapping*), Projekt 2 będzie stanowił *Execution unit*, z RST jako *External variable*.

Sprawdzenia, czy obydwie jednostki wykonawcze zostały połączone można dokonać za pomocą symulatorów CPSim (pkt. 3.4) pod warunkiem, że WinController został wybrany jako źródło danych. W oknie CPDev powinny się wtedy znajdować obydwa rozpatrywane projekty, jak w drzewie po lewej stronie rysunku 6.6, po czym po aktywacji symulacji *online* należy wskazać jednostkę wykonawczą dla każdego z nich. Wówczas symulatory CPSim zostaną połączone z WinControllerem przekazującym zmienne odpowiednio do *I/O mapping*. Po utworzeniu paneli wizualizacyjnych w CPSim można śledzić i ustawiać odpowiednie zmienne.



Rys. 6.6. Okno CPDev oraz dwa okna CPSim dla projektów wykonywanych przez WinController.

Po prawej stronie rysunku 6.6 pokazano okna symulatorów CPSim z WinControllerem po pięciu kolejnych impulsach na wejściu IN1 (CNT=5 w obydwu oknach). Ponieważ wartość 5 przekracza próg 3 więc wyjście OUT2 w Projekcie 2 jest ustawione.

W sumie więc, WinController może służyć do symulacji współpracy rdzeni przed faktyczną implementacją sprzętową.

Warto zaznaczyć, że WinController pozwala symulować projekty wykonywane z różnymi cyklami, czyli takie jak na rysunkach 6.4a,b (10 i 50 ms). W sytuacji jednak gdyby obydwa cykle były jednakowe, to do symulacji wspólnego projektu z rysunku 6.3 wystarczyłby symulator CPSim lub środowisko IDE (obszerniejsze). Sytuacji takiej dotyczy system przedstawiony w następnym rozdziale.

6.4. Obsługa pamięci współdzielonej

Maszyny wirtualne w każdym z rdzeni procesora powinny realizować kopiowanie wymienianych zmiennych globalnych ze swoich pamięci danych do pamięci współdzielonej lub odwrotnie. Oczywiście kopiowanie takie może być prowadzone przez daną maszynę tylko wtedy, gdy druga maszyna akurat tego nie wykonuje. Rozszerzenie języka pośredniego VMASM na kopiowanie lokalnej pamięci danych DM do/z pamięci współdzielonej SM obejmuje dwie procedury systemowe (instrukcje):

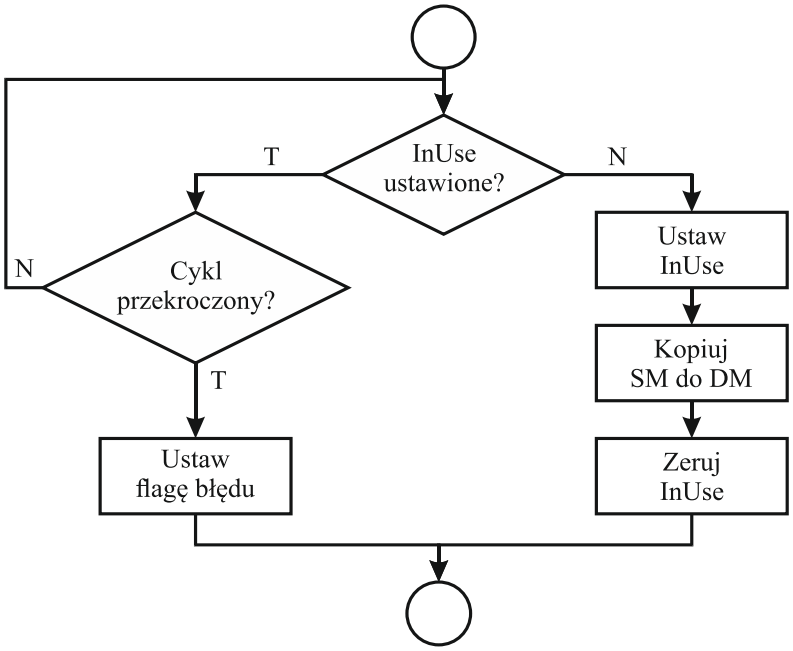
- `SM_TO_DM (SharedAddress, LocalAddress, ByteNumber)`
- `DM_TO_SM (LocalAddress, SharedAddress, ByteNumber)`

Adresy oraz liczba bajtów są nadawane przez kompilator. Procedura SM_TO_DM jest wykonywana w fazie *precycle* maszyny wirtualnej w odniesieniu do zmiennych z atrybutem READ. Jak podano w punkcie 3.5, zmienne te są dalej kopiowane do wewnętrznych kopii w maszynie, które faktycznie służą do obliczeń. Natomiast procedura DM_TO_SM jest wykonywana w fazie *postcycle* dla zmiennych z atrybutem WRITE, które później odczytuje druga maszyna.

Należy jeszcze zwrócić uwagę, że gdy wymieniane zmienne są zadeklarowane w dwóch grupach odpowiednio do atrybutów, tzn. jedna grupa zmiennych z READ, a druga z WRITE, to procedury SM_TO_DM i DM_TO_SM wykonywane są jednorazowo we właściwej fazie. Natomiast w przypadku, gdy brak takiego uprządkowania, procedury będą powtarzane odpowiednio do liczby grup zmiennych mających jednakowe atrybuty.

Procedury dotyczące danego rdzenia dokonują faktycznego kopiowania tylko wtedy, gdy systemowa flaga nazwana tutaj InUse, a sygnalizująca aktualne korzystanie z pamięci współdzielonej, nie została wcześniej ustawiona (zajęta) przez drugi rdzeń. W przeciwnym razie procedura oczekuje na wyzerowanie flagi (zwolnienie) i dopiero wtedy kopiuje. W związku z tym algorytm procedury SM_TO_DM ma postać jak na rysunku 6.7. Flaga błędu ustawiana w razie przekroczenia czasu cyklu aktywuje obsługę błędu przez interfejs platformy

docelowej maszyny (pkt. 3.1). Można dodać, że proponowany dla CPDev sposób obsługi błędów przedstawiono niedawno w [55]. W przypadku procedury DM_TO_SM algorytm różniłby się krokiem *Kopiuj DM do SM*.



Rys. 6.7. Algorytm procedury kopiowania SM_TO_DM.

W systemach mikrokontrolerowych mechanizm oparty na pojedynczej fladze jest często wystarczający, ponieważ brak systemu operacyjnego ogranicza możliwość współbieżnego dostępu do wspólnych zasobów. Natomiast na platformach mikroprocesorowych pracujących w środowisku wielozadaniowym stosuje się bardziej rozbudowane mechanizmy synchronizacji (pkt. 5.4). Mechanizmy te zapewniają, że tylko jeden wątek w danym momencie może uzyskać dostęp do wspólnego zasobu.

W konkretnej implementacji flaga InUse jest utożsamiana z semaforem sprzętowym danego procesora wielordzeniowego lub semaforem programowym systemu operacyjnego. Przykładowo, mikrokontroler STM32H755 implementuje semafory sprzętowe zwane HSEM [38], a mikrokontroler RP2040 udostępnia nieco prostszy mechanizm Spinlock [28]. Na listingu 6.1 pokazano podstawowy kod z semaforem HSEM o numerze `HSEM_ID` realizującym procedurę SM_TO_DM, który odpowiada algorytmowi graficznemu z rysunku 6.7 z pomięciem badania cyklu i flagi błędu. Implementację przedstawiono w koncepcji aktywnego czekania (*busy-waiting*). Flaga InUse jest reprezentowana przez porównanie z `HAL_OK`, a oczekiwanie na zwolnienie semafora przez pętlę `while`. Funkcje `HAL_HSEM_Take` i `HAL_HSEM_Release` reprezentują odpowiednio ustawienie (zajęcie) i zwolnienie semafora. Funkcje te przyjmują jako argumenty numer semafora `HSEM_ID` identyfikujący współdzielony

zasób oraz numer rdzenia (lub procesu) `CORE_ID`, który zajmuje semafor. Dzięki temu możliwa jest poprawna synchronizacja pamięci współdzielonej między rdzeniami.

```
while(HAL_HSEM_Take(HSEM_ID, CORE_ID) != HAL_OK)
  { }
CopySharedMemToDataMem();
HAL_HSEM_Release(HSEM_ID, CORE_ID);
```

Listing 6.1. Kod procedury SM_TO_DM z semaforem sprzętowym.

6.5. Rozszerzenie modelu denotacyjnego

Przedstawione niżej rozszerzenie bazuje na dziedzinach, funkcjach i modelach wprowadzonych w punkcie 3.3 i w Dodatku A. W szczególności pamięć współdzieloną definiuje się jako alias dziedziny *Memory*, czyli

$$SharedMemory = Memory \tag{6.1}$$

Stan systemu zawierającego dwie maszyny wirtualne na osobnych rdzeniach oraz pamięć współdzieloną jest określony jako

$$\begin{aligned} CoreState = {} & CodeMemory \times DataMemory \times CodeStack \times \\ & \times DataStack \times CodeReg \times DataReg \times Flags \\ State = {} & CoreState \times CoreState \times SharedMemory \times SharedFlags \end{aligned} \tag{6.2}$$

z dodatkowymi flagami statusowymi $SharedFlags$. Oznaczając obydwie maszyny za pomocą indeksów A, B, wartości składające się na aktualny stan można przedstawić za pomocą krotek, tj.

$$\begin{aligned} ((cm_A, dm_A, cs_A, ds_A, cr_A, dr_A, flg_A), \\ (cm_B, dm_B, cs_B, ds_B, cr_B, dr_B, flg_B), sm, sflg) \end{aligned} \tag{6.3}$$

Nowością w rozszerzonym modelu są procedury systemowe DM_TO_SM i SM_TO_DM realizujące zapis i odczyt do pamięci współdzielonej. Niżej przedstawiono model denotacyjny pierwszej z nich przy założeniu, że zapis do pamięci współdzielonej realizuje maszyna A. Struktura modelu nawiązuje wprost do rysunku 6.7 i kodu z listingu 6.1, z tym że teraz zamiast semafora HSEM użyto oryginalnej flagi InUse, a do przekroczenia czasu cyklu flagi TimeUp (należącej do flg_A). Aktualizację pamięci współdzielonej sm przez maszynę A zdefiniowano jako podstawienie usm wykorzystujące funkcję dziedzinową *MemMove* (Dodatek A). Oczekiwanie *while* na zwolnienie flagi InUse ma postać stałopunktowego równania denotacyjnego [32,33] ze względu na *while* po lewej i prawej stronie. Równanie takie reprezentuje proces iteracyjny. Pozostałe instrukcje modelu są takie

same jak w przykładach z punktu 3.3 i w Dodatku A. Można jeszcze przypomnieć, że cr_3 zawiera adres następnej instrukcji po DM_TO_SM (pkt. 3.3).

$$
\begin{aligned}
&\mathcal{C}[\![DM\_TO\_SH\colon LocalAddress\colon SharedAddress\colon ByteNumber]\!] = \\
&\qquad = \lambda s.\begin{pmatrix}(cm_A, dm_A, cs_A, ds_A, cr_A, dr_A, flg_A),\\ (cm_B, dm_B, cs_B, ds_B, cr_B, dr_B, flg_B),\\ sm, sflg\end{pmatrix} := s \\
&srcaddr := GetAddress(cr_A, cm_A) \\
&cr_1 := cr_A \oplus AddressSize \\
&dstaddr := GetAddress(cr_1, cm_A) \\
&cr_2 := cr_1 \oplus AddressSize \\
&size := ByteOf\big(Get1BMem(cr_2, cm_A)\big) \\
&cr_3 := cr_2 \oplus 1 \qquad (6.4)\\
&\{usm := MemMove(dm_A, srcaddr, sm, dstaddr, size)\} \\
&\textbf{\textit{while}}\ not\ InUse\ \textbf{\textit{do}}\ usm\ = \\
&\quad \textbf{\textit{if}}\ not\ InUse\ \textbf{\textit{then}}\ usm \\
&\quad \textbf{\textit{else}}\ (\textbf{\textit{if}}\ TimeUp\ \textbf{\textit{then}}\ error \\
&\qquad \textbf{\textit{else while}}\ not\ InUse\ \textbf{\textit{do}}\ usm) \\
&s1 := \begin{pmatrix}(cm_A, dm_A, cs_A, ds_A, cr_3, dr_A, flg_A),\\ (cm_B, dm_B, cs_B, ds_B, cr_B, dr_B, flg_B),\\ usm, sflg\end{pmatrix} \\
&s1
\end{aligned}
$$

Model procedury SM_TO_DM wygląda podobnie, z tym że zamiast usm podstawieniem jest

$$udm_A := MemMove(sm, srcaddr, dm_A, dstaddr, size) \qquad (6.5)$$

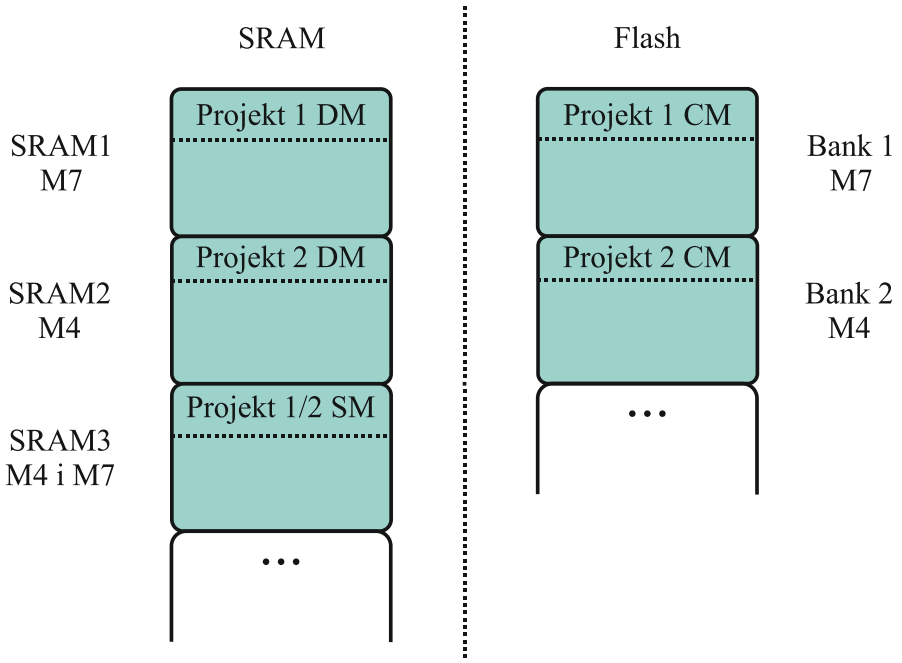
z odpowiednią zmianą w określeniu nowego stanu s_1.

W sumie na model denotacyjny systemu dwurdzeniowego składają się dziedziny i funkcje dziedzinowe podane w Dodatku A, uniwersalna funkcja $\mathcal{U}$ z punktu 3.3 wybierająca instrukcje VMASM, modele tych instrukcji jak w przedstawionych przykładach oraz powyższe modele kopiowania do/z pamięci współdzielonej.

6.6. Implementacja w mikrokontrolerze STM32

Spośród wymienionych w punkcie 5.2 platform wykonawczych zdecydowano się na rozbudowany mikrokontroler STM32H755ZIT6 [38] składający się z dwóch rdzeni Cortex-M4 i Cortex-M7. Odmienna charakterystyka rdzeni rozszerza zakres zastosowań. Ważnym aspektem przy wyborze były także szerokie zasoby pamięci i wbudowane peryferia.

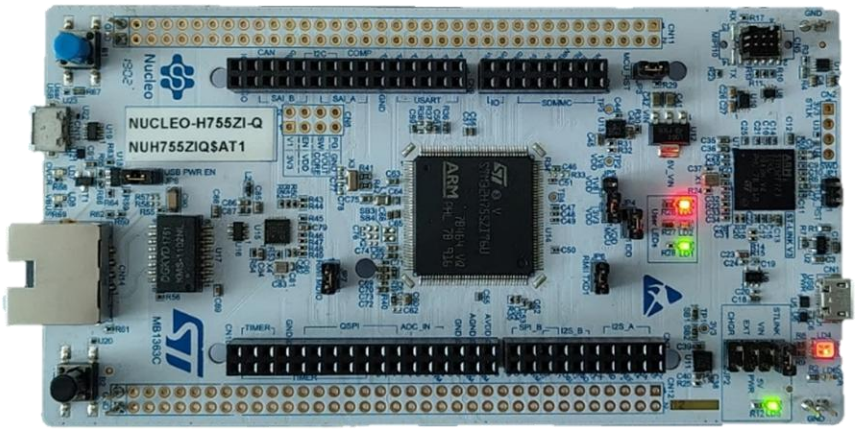
Wymiana zmiennych globalnych wymaga pamięci współdzielonej dostępnej z każdego z rdzeni. Układy z serii STM32H7 posiadają wbudowane dwa rodzaje pamięci nadające się do tego celu, tj. standardową SRAM (*Static Random Access Memory*) i AXI SRAM (*Advanced eXtensible Interface SRAM*). W przypadku pamięci SRAM część sektorów dostępna jest z poziomu obydwu rdzeni, może więc służyć jako pamięć współdzielona. Dostęp równoległy do wspólnego obszaru pamięci wymaga naturalnie synchronizacji. Układy takie jak STM32H755/745 oraz STM32H757/747 wyposażone zostały we wspomniane sprzętowe semafory HSEM, których cechą jest mechanizm atomowego przejmowania semafora, co pozwala uniknąć stosowania systemu operacyjnego do synchronizacji dostępu. Wybór typowych rodzajów pamięci i synchronizacji semaforowej pozwala na utrzymanie jednolitej struktury oprogramowania między różnymi platformami sprzętowymi i systemowymi. Diagram na rysunku 6.8 prezentuje uproszczony podział wykorzystywanych sektorów pamięci danych DM, współdzielonej SM i pamięci kodu CM dla rdzeni, które mają do nich dostęp.



Rys. 6.8. Uproszczony diagram podziału pamięci SRAM i Flash w układzie STM32H755.

Do uruchomienia przykładowych projektów z punktu 6.1 zastosowano pokazaną na rysunku 6.9 płytę uruchomieniową NUCLEO_H755ZI-Q z układem STM32H755ZIT6U firmy STMicroelectronics. Do interakcji z programami wykorzystano monostabilny przycisk B1 (niebieski, lewy górny róg) i diody LED L1, L3 (czerwona i zielona, pośrodku płyty). Wejście IN1 z rysunku 6.1a zostało powiązane z przyciskiem B1, a wyjścia OUT1, OUT2 z diodami, tzn. OUT1 z L1 a OUT2 z L3. Do obsługi wejść i wyjść wykorzystano funkcje interfejsu platformy docelowej `WM_GetData` oraz `WM_SetData`. Stan płyty pokazany na rysunku 6.9

występuje po dziesięciu naciśnięciach przycisku B1, kiedy wyjścia OUT1 i OUT2 są ustawione, a więc diody świecą. Kontrolowanie stanu wejścia RST, śledzenie aktualnego stanu obydwu rdzeni, szczególnie parametrów każdej z maszyn wirtualnych CPDev i aktualnej wartości zmiennej CNT, przeprowadzono w trybie debugowania za pomocą wbudowanego programatora i debugera ST-LINK w środowisku STM32CubeIDE.



Rys. 6.9. Płyta uruchomieniowa NUCLEO_H755ZI-Q z uruchomionymi Projektami 1, 2.

W domyślnej, dwurdzeniowej konfiguracji układu STM32H755ZIT6U, inicjacja startowa przeprowadzana jest przez Cortex-M7 (CM7). Po włączeniu zasilania, startujący podrzędny rdzeń Cortex-M4 (CM4) konfiguruje blok sprzętowy semaforów HSEM i aktywuje powiadomienie o zmianie stanu semafora, po czym przechodzi w *STOP mode*. Równolegle, nadrzędny CM7 oczekuje na dezaktywację CM4 (uśpienie – *sleep*). Po zakończeniu dezaktywacji CM4 rdzeń CM7 dokonuje bazowej konfiguracji systemu (zegary, pętle PLL, pamięć, peryferia). Po zakończeniu konfiguracji, CM7 poprzez HSEM aktywuje CM4 (wybudzenie – *wake-up*), który po uruchomieniu zeruje flagi semaforów i przechodzi do dalszej części oprogramowania. Dzięki temu mechanizmowi rdzenie CM4 i CM7 w zasadzie jednocześnie uruchamiają wykonywanie głównego oprogramowania, rozpoczynając od inicjacji przypisanych peryferii. Ostatnim etapem przed cykliczną pracą jest przypisanie właściwego kodu binarnego do maszyny wirtualnej CPDev. Uruchomienie działania maszyny wirtualnej przeprowadza się funkcją `WM_Initialize` z domyślnymi parametrami `WM_MODE_FIRST_START | WM_MODE_NORMAL`, powodując ustawienie początkowych wartości zmiennych i start w trybie normalnym.

Bazowa implementacja na układzie STM32H755 składa się z kilku uzupełniających elementów. Do synchronizacji wykonywania zadań przez rdzenie służy podstawa czasu systemowego oparta o przerwania licznika sprzętowego SysTick. Dostęp do współdzielonych

peryferii, takich jak GPIO, odbydwa się w logice przynależności zapisu/edycji przez wybrany rdzeń, podobnie jak ma to miejsce przy wymianie zmiennych globalnych. Do komunikacji z układami peryferyjnymi wykorzystano bibliotekę HAL (*Hardware Abstraction Layer*), która zwiększa przenośność i czytelność kodu.

Implementacja działania sterownika przedstawiona na listingu 6.2 rozpoczyna się od sprawdzenia flagi `bRunMode` zezwalającej na pracę maszyny wirtualnej. Następnie, na podstawie aktualnego czasu systemowego odczytywanego za pomocą funkcji `HAL_GetTick` z dokładnością do 1 ms, kontrolowany jest moment uruchomienia cyklu `tCycle`. Etap *precycle*, czyli przygotowanie sterownika do obliczeń (odpowiednik funkcji `VMP_PreCycle` z pkt. 3.5) rozpoczyna się od wywołania funkcji `inputUpdate`. W tym miejscu aktualizowane są globalne zmienne wejściowe, np. poprzez zapis stanów pinów wejściowych mikrokontrolera w pamięci danych maszyny wirtualnej.

```
    while(bRunMode){
      if (HAL_GetTick() - tCycleStart > tCycle){
        // precycle
        tCycleStart = HAL_GetTick();
        inputUpdate();
        // SM_TO_DM
        while (HAL_HSEM_Take(HSEM_ID, CORE_ID) != HAL_OK)
          { }
        copySharedMemToDataMem();
        HAL_HSEM_Release(HSEM_ID, CORE_ID);
        // cycle
        runCycle();
        // postcycle
        outputUpdate();
        // DM_TO_SM
        while (HAL_HSEM_Take(HSEM_ID, CORE_ID) != HAL_OK)
          { }
        copyDataMemToSharedMem();
        HAL_HSEM_Release(HSEM_ID, CORE_ID);
        // cycle overflow detection
        if (HAL_GetTick() - tCycleStart > tCycle)
          wStatus |= TIME_CYCLE_OVERFLOW;
      }
    }
```

Listing 6.2. Uproszczony fragment implementacji maszyny wirtualnej CPDev na jednym z rdzeni z synchronizacją opartą o semafor sprzętowy HSEM [53].

Kolejnym krokiem jest uzyskanie dostępu do pamięci współdzielonej przez pobranie semafora o numerze `HSEM_ID` (funkcja `HAL_HSEM_Take`), a następnie pobranie wartości wymienianych zmiennych globalnych z atrybutem READ (funkcja

`copySharedMemToDataMem`). Właściwe obliczenia maszyny wirtualnej realizowane są w funkcji `runCycle`. Ostatni etap cyklu, czyli *postcycle* będący odpowiednikiem `VMP_PostCycle` (pkt. 3.5) polega na aktualizacji wyjść funkcją `outputUpdate` oraz zmiennych globalnych z atrybutem WRITE, analogicznie do etapu *precycle*. Cykl kończy się sprawdzeniem i warunkowym zapisaniem faktu przekroczenia zadanego czasu cyklu przechowywanego w zmiennej systemowej `tCycle`.

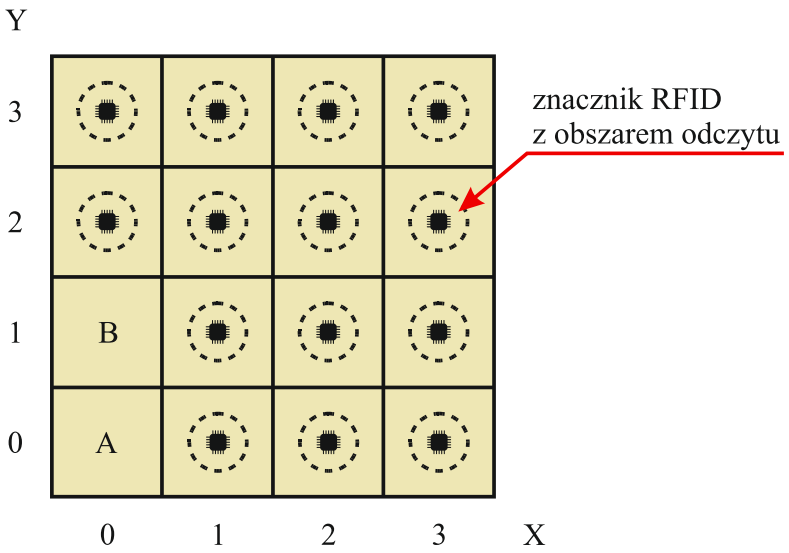
7. Laboratoryjny wielordzeniowy system eksploracji RFID

W rozdziale przedstawiono bardziej złożony system z dwoma odmiennymi jednostkami sterującymi, w którym chodzi o identyfikację znaczników RFID rozmieszczonych na płycie w siatce kwadratowej. Odczyt znaczników przeprowadza mały robot mobilny z czytnikiem RFID, sterowany przez dwurdzeniowy mikrokontroler ESP32. Przemieszczaniem robota, czyli eksploracją płyty steruje jednostka nadrzędna w postaci czterordzeniowego mikrokomputera jednopłytkowego Raspberry Pi, komunikująca się z ESP32 poprzez Wi-Fi. Pierwotna wersja systemu, ale ze specjalnie skonstruowanymi mikrorobotami klasy (2,0) i oprogramowaniem napisanym w C/C++ została opisana w [42,81]. Dwa rdzenie Raspberry Pi z maszynami wirtualnymi CPDev1, CPDev2 kontrolują eksplorację i poprzez Wi-Fi wysyłają komendy dotyczące ruchu robota do maszyny CPDev3 w jednym z rdzeni ESP32. Wykonywanie tych komend w formie sterowania silnikami prowadzi dodatkowe oprogramowanie umieszczone w drugim rdzeniu ESP32. Dwa następne rdzenie Raspberry Pi za pośrednictwem systemu Linux obsługują interfejsy maszyn wirtualnych, komunikację Wi-Fi oraz realizują funkcje systemowe. Oprogramowanie sterujące całością zostało utworzone w języku ST jako jeden wspólny projekt, z którego podczas implementacji wydzielono podprojekty dla maszyn CPDev1, CPDev2 i CPDev3. Wymiana zmiennych globalnych CPDev1 z CPDev2 w Raspberry Pi odbywa się poprzez pamięć współdzieloną, a CPDev2 z CPDev3 oraz CPDev3 z CPDev1 poprzez Wi-Fi, ale z zachowaniem tego samego mechanizmu (pkt. 5.3). Podano listę wymienianych zmiennych globalnych, wykaz jednostek POU we wspólnym projekcie oraz wyniki wstępnej symulacji. Kody programów POU znajdują się w Dodatku B. W ramach implementacji laboratoryjnej, oprócz przekonstruowania napędu robota i podziale wspólnego projektu na maszyny CPDev, dotychczasowe firmowe oprogramowanie robota sterujące ruchem zastąpiono własnym oprogramowaniem z regulatorami PID i bang-bang. Pomimo tego, wobec braku systemu lokalizacji położenia typu GNSS RTK, na skutek różnych niedokładności i zakłóceń, robot zbacza jednak stopniowo z wyznaczonej trasy i nie napotykając oczekiwanego znacznika RFID zatrzymuje się sygnalizując pominięcie. Należy wtedy ustawić robota nad ostatnio zidentyfikowanym znacznikiem i kontynuować eksplorację.

7.1. Problem identyfikacji RFID w obszarze

Załóżmy, że pewien prostokątny obszar został podzielony siatką kwadratową jak na rysunku 7.1, w której polach umieszczono znaczniki RFID (transpondery) [82]. Alternatywą mogą być znaczniki Bluetooth (tzw. *beacon*) lub kody QR (*Quick Response*). Za współrzędne

X,Y danego pola uważa się numery odpowiadającej mu kolumny i wiersza siatki. W sytuacji pokazanej na rysunku na polu [0,0] znajduje się znacznik z odczytanym identyfikatorem A, a na polu [0,1] znacznik z identyfikatorem B. Identyfikatory na pozostałych polach nie zostały jeszcze odczytane. Problemem jest więc określenie jakie identyfikatory mają pozostałe znaczniki, co nazywa się rozpoznaniem obszaru albo eksploracją.



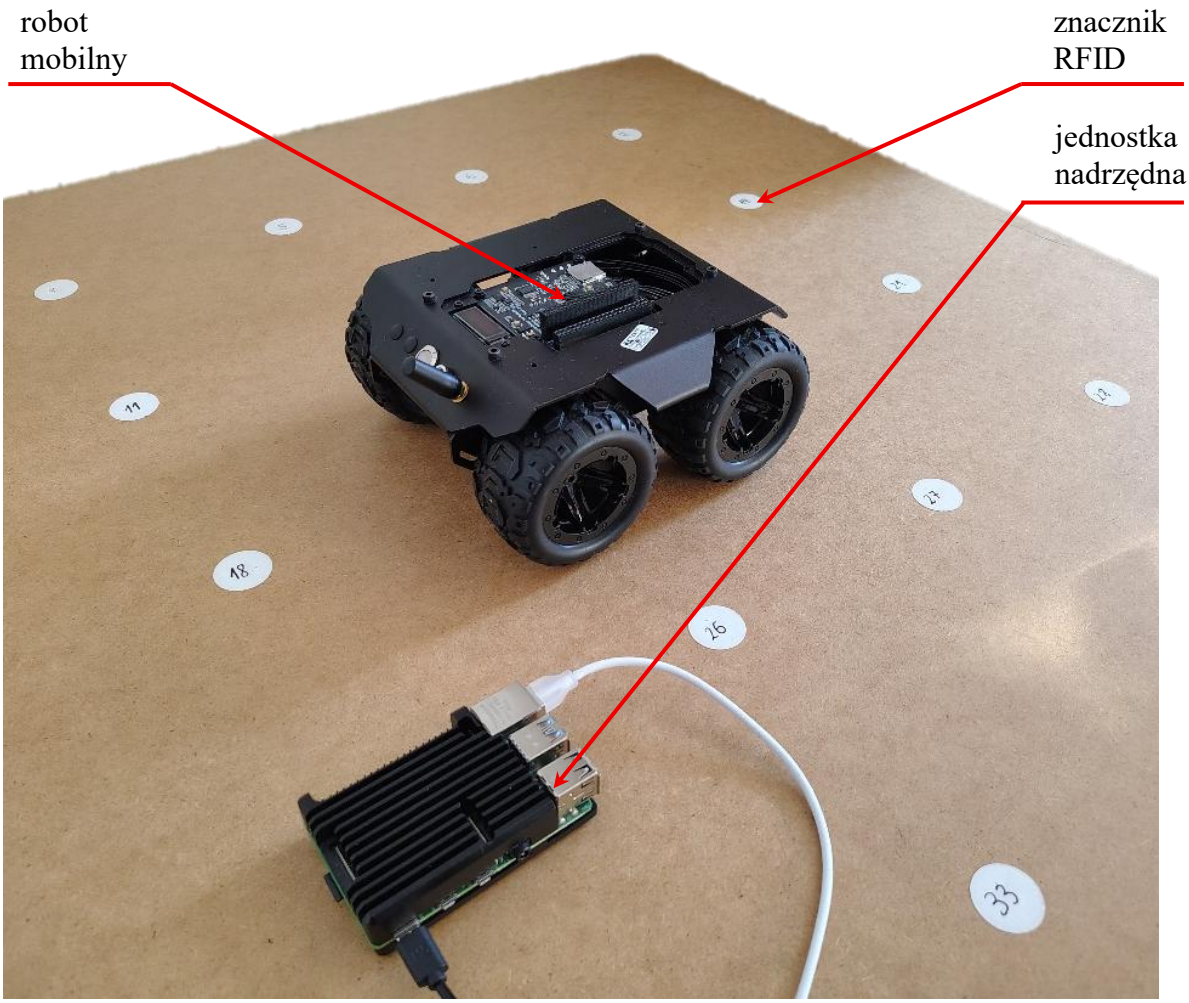
Rys. 7.1. Przykład obszaru eksploracji RFID.

Problem tego typu występuje podczas przygotowywania infrastruktury RFID na powierzchni hali przeznaczonej dla robotów mobilnych. Bezpośrednio po montażu znaczników w wyznaczonych miejscach ich identyfikatory RFID nie są znane, bądź zachodzi potrzeba ich eksperymentalnej weryfikacji. Dopiero po zidentyfikowaniu hali roboty są w stanie podejmować zadania nawigacyjne w oparciu o znaczniki.

Eksplorację obszaru można przeprowadzić ręcznie korzystając z czytnika RFID, albo automatycznie za pomocą robota mobilnego wyposażonego w czytnik oraz układ lokalizacji położenia typu GNSS RTK [83] wskazujący gdzie są znaczniki. Opracowaną laboratoryjną wersję takiego systemu, który nie ma układu lokalizacji położenia, pokazano na rysunku 7.2, gdzie obszarem eksploracji jest płyta ze znacznikami RFID, a za jednostkę wykonawczą służy mały robot mobilny komunikujący się bezprzewodowo (Wi-Fi) z mikrokomputerową jednostką nadrzędną [42,81]. Organizacja eksploracji, komunikacja i sterowanie robotem są podstawowymi zadaniami systemu.

Spośród szeregu możliwych algorytmów eksploracji wybrano algorytm z punktem końcowym w środku obszaru. Wymaga to rozpoczęcia eksploracji w jednym z naroży i podążania na przemian w kierunku X lub Y z obrotami o 90^0 tylko w jedną stronę. Przykładową trasę rozpoczynającą się w polu [0,0] i obrotami tylko w prawo pokazano na

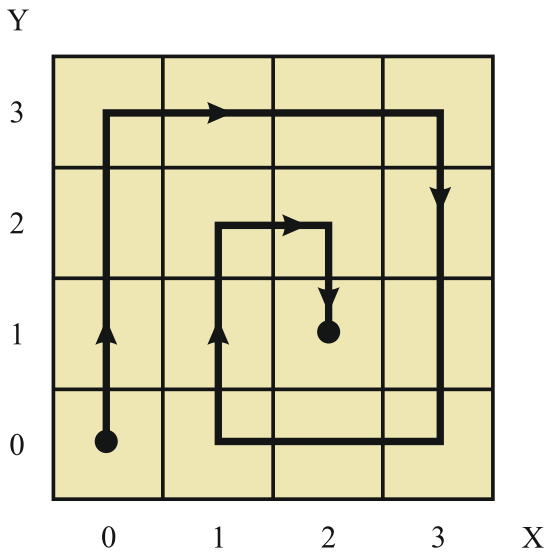
rysunku 7.3. Ścieżki biegnące na wprost kończą się na granicy obszaru lub po napotkaniu zidentyfikowanego pola. Wtedy następuje obrót w prawo i ewentualna jazda na wprost, o ile prawe pole nie zostało jeszcze zidentyfikowane. Jeżeli pola na wprost i po prawej stronie zostały już zidentyfikowane, eksploracja kończy się, jak w polu [2,1] na rysunku 7.3.



Rys. 7.2. Urządzenia laboratoryjnego systemu eksploracji RFID.

Przemysłowa realizacja powyższego algorytmu wymagałaby wyposażenia robota we wspomniany system lokalizacji położenia wraz z mapą, gdzie należy szukać znaczników. Tymczasem robot wykorzystywany w przedstawionym, uproszczonym rozwiązaniu dysponuje jedynie siatką kwadratową, ale bez możliwości weryfikacji swego położenia względem umieszczonych na niej znaczników. Należy więc się spodziewać, że na skutek różnych niedokładności i zakłóceń, odchylenie robota od planowej trasy stanie się po pewnym czasie na tyle duże, że po przemieszczeniu przekraczającym nieco rozmiar siatki robot nie napotka już oczekiwanego znacznika. W odniesieniu do takiej sytuacji przyjęto więc, że nie napotkawszy na oczekiwany znacznik robot powinien się zatrzymać i sygnalizować potrzebę interwencji. Interwencja taka powinna polegać na umieszczeniu robota nad ostatnim zidentyfikowanym

znacznikiem na wprost ścieżki i poleceniu kontynuacji eksploracji. W ten sposób należy zatem rozumieć postępowanie w jej laboratoryjnej realizacji.



Rys. 7.3. Eksploracja z punktem końcowym pośrodku obszaru.

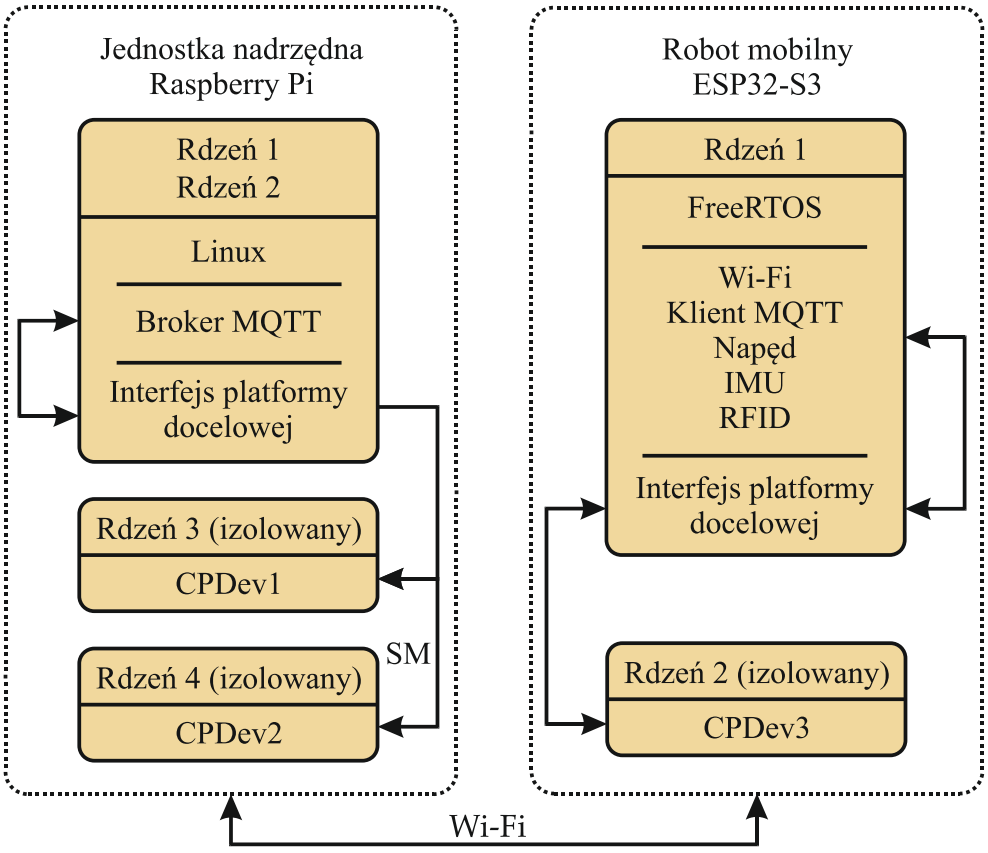
7.2. Architektura rozwiązania z robotem i jednostką nadrzędną

Przedstawiany system wykorzystuje zróżnicowaną – heterogeniczną architekturę, w której jednostka nadrzędna oparta na czterordzeniowym jednopłytkowym mikrokomputerze Raspberry Pi kontroluje pracę jednostki podrzędnej, czyli robota mobilnego wyposażonego w dwurdzeniowy mikrokontroler ESP32-S3. Podział zasobów obydwu jednostek i ich głównych zadań pokazano na rysunku 7.4. Podstawę implementacji programowej stanowią trzy maszyny wirtualne CPDev z wymianą zmiennych globalnych różnymi kanałami komunikacyjnymi, tzn. poprzez pamięć współdzieloną SM oraz bezprzewodowo poprzez Wi-Fi. Numeracja maszyn wirtualnych na rysunku odpowiada kolejności operacji w procesie eksploracji.

Jednostka nadrzędna pełni rolę centralnego węzła przetwarzania danych i wyznaczania sterowania. Raspberry Pi wyposażono w czterordzeniowy procesor Broadcom BCM2711 z rdzeniami Cortex-A72 (64-bit) o taktowaniu do 1,5 GHz. Oprogramowanie oparte o system operacyjny Linux (ściślej Raspberry Pi OS) umożliwia zarządzanie procesami sterowania oraz obsługę komunikacji Wi-Fi. Dwie maszyny wirtualne uruchamiane są w izolowanych rdzeniach, niedostępnych dla procesów systemu operacyjnego. Maszyna CPDev1 odpowiada za rejestrację odczytywanych znaczników RFID w tablicy reprezentującej rozpoznawaną siatkę, natomiast CPDev2 zajmuje się wyznaczaniem komend sterujących robotem zgodnie z algorytmem eksploracji (koniec w środku obszaru). Zastosowanie systemu operacyjnego

zapewnia sterowniki i oprogramowanie integrujące komunikację z interfejsami maszyn wirtualnych.

Robot mobilny działający jako podrzędna jednostka wykonawcza zawiera płytę sterującą General Driver Board for Robots firmy Waveshare [40] z mikrokontrolerem ESP32-S3. Zgodnie z zaleceniami producenta układu Espressif, dla poprawnej pracy oprogramowania i komunikacji Wi-Fi, wskazane jest zastosowanie systemu operacyjnego FreeRTOS. Dwurdzeniowa architektura ESP32-S3 wykorzystująca 32-bitowe rdzenie Xtensa LX7 [60] pozwala na rozdzielenie części sterującej z maszyną wirtualną CPDev3 od obsługi pozostałych elementów płyty i interfejsu platformy docelowej (rys. 7.4).



Rys. 7.4. Zasoby systemu eksploracji RFID i przeznaczenie rdzeni.

Maszyna CPDev3 odpowiada za wykonywanie przez robota komend otrzymywanych z CPDev2 oraz za pośrednictwem oprogramowania niskopoziomowego, zwracanie do CPDev1 nowo odczytanego identyfikatora RFID. Komunikacja Wi-Fi pomiędzy jednostkami realizowana jest za pośrednictwem protokołu MQTT, który pozwala na rozpowszechnianie informacji do modyfikowalnej puli odbiorców, dzięki czemu można monitorować działanie systemu z poziomu zewnętrznych narzędzi.

Niskopoziomowe oprogramowanie robota implementuje otrzymaną komendę, prowadzi komunikację Wi-Fi oraz obsługuje sytuację nienormalną, tzn. gdy pomimo

wykonania ruchu zleconego przez komendę, na skutek kumulujących się odchyleń, robot pominął oczekiwany znacznik. Jak podano poprzednio, sygnalizowana jest wtedy potrzeba interwencji, w wyniku której robot powtarza ostatni ruch. Dopiero po jego wykonaniu odczytany identyfikator RFID jest przekazywany do CPDev1. Z punktu widzenia maszyn wirtualnych CPDev obsługujących normalne funkcjonowanie systemu, sytuacja nienormalna oznacza dłuższe oczekiwanie na nowy identyfikator RFID.

7.3. Współpraca trzech maszyn wirtualnych

Omawiana współpraca dotyczy sytuacji normalnej, tzn. gdy w wyniku wykonania odpowiedniej komendy zostaje odczytany identyfikator RFID danego znacznika. Opisana wyżej sytuacja nienormalna jest scharakteryzowana bliżej w ramach implementacji (pkt. 7.5).

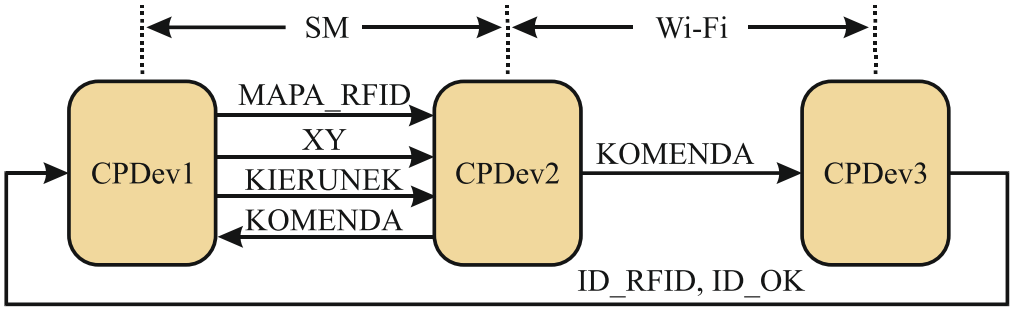
Podstawowe zmienne globalne, z których korzystają maszyny wirtualne podano w tabeli 7.1. Zmienne, gdzie po prawej stronie występują READ i WRITE są zmiennymi wymienianymi. Pierwsza z tych zmiennych – dwuwymiarowa tablica MAPA_RFID zawiera identyfikatory na kolejnych polach rozpoznawanej siatki lub wartość 0 tam, gdzie znaczników jeszcze nie zidentyfikowano. KIERUNEK ruchu może być zgodny lub przeciwny w stosunku do osi X lub Y. Para XY reprezentuje współrzędne pola siatki, czyli elementu tablicy. KOMENDA jest interpretowana jako NA_WPROST, OBR_LEWO, OBR_PRAWO i STOP, zgodnie z oznaczeniami użytymi w następnym punkcie i w Dodatku B. ID_RFID jest identyfikatorem znacznika na nowo zidentyfikowanym polu, a poprawność jego odczytu potwierdza flaga ID_OK poprzez TRUE. Flagę tę po pewnym czasie robot ustawia na FALSE sygnalizując gotowość do następnego odczytu.

Tab. 7.1. Wymieniane zmienne globalne i parametry systemu eksploracji RFID.

Zmienna	**Typ**	**CPDev1**	**CPDev2**	**CPDev3**
MAPA_RFID	ARRAY […,…] OF DWORD	WRITE	READ	-
KIERUNEK	INT	WRITE	READ	-
XY	ARRAY [0..1] OF INT	WRITE	READ	-
KOMENDA	INT	READ	WRITE	READ
ID_RFID	DWORD	READ	-	WRITE
ID_OK	BOOL	READ	-	WRITE
ROBOT_POLECENIE	INT	-	-	WRITE
ROBOT_WARTOSC	INT	-	-	WRITE
OBROT	WORD	-	READ	-
MAPA_ROZMIAR	INT	-	-	READ
MAPA_ROZSTAW	INT	-	-	READ

Pozostałe zmienne globalne w tabeli 7.1 są podstawowymi parametrami systemu (READ) lub danymi dla oprogramowania niskopoziomowego (WRITE) przekazywanymi poprzez interfejs. W szczególności ROBOT_POLECENIE i ROBOT_WARTOSC implementują komendę w układzie wykonawczym robota. OBROT jako LEWO lub PRAWO oznacza stronę, w którą robot może się obracać podczas całego procesu eksploracji. MAPA_ROZMIAR reprezentuje współrzędną ostatniego wiersza lub kolumny (licząc od 0), zatem tablica 4×4 przedstawiona na rysunku 7.3 reprezentowana będzie przez wartość 3. MAPA_ROZSTAW oznacza odległość między znacznikami.

Odpowiednio do atrybutów READ/WRITE wymianę zmiennych globalnych między maszynami wirtualnymi ilustruje rysunek 7.5, który w odniesieniu do identyfikacji jednego znacznika można zinterpretować następująco. Po otrzymaniu z CPDev2 wykonywanej komendy, a z CPDev3 identyfikatora ID_RFID i potwierdzenia ID_OK (flagi), maszyna CPDev1 w zależności od wartości zmiennej KOMENDA aktualizuje tablicę MAPA_RFID lub tylko KIERUNEK. Jeżeli wartość komendy była NA_WPROST, to ID_RFID jako nowo odczytany identyfikator zostaje wpisany do MAPA_RFID na nowej pozycji XY. Natomiast gdy komendą było OBR_LEWO lub OBR_PRAWO i robot wykonał obrót w zidentyfikowanym wcześniej miejscu XY, to aktualizowany jest tylko KIERUNEK dalszej eksploracji, a MAPA_RFID pozostaje niezmieniona.



Rys. 7.5. Współpraca maszyn wirtualnych poprzez wymianę zmiennych globalnych.

Maszyna CPDev2 na podstawie zmiennych MAPA_RFID, XY i KIERUNEK generuje nową komendę odpowiednio do schematu eksploracji (rys. 7.3). Jeżeli pole na wprost robota jest niezidentyfikowane (PUSTE), to nową komendą jest NA_WPROST. W przypadku jednak, gdy pole na wprost zostało już zidentyfikowane, maszyna CPDev2 bada, czy pole boczne po stronie OBROT jest niezidentyfikowane. Jeżeli tak, to nową komendą jest OBR_LEWO lub OBR_PRAWO. Natomiast, gdy pole boczne po stronie OBROT okazuje się zidentyfikowane, CPDev2 generuję komendę STOP, bo oznacza to, że robot dotarł do środka eksplorowanej siatki.

Maszyna CPDev3 pośredniczy między częścią decyzyjną systemu, a układem wykonawczym robota zastępując komendę zmiennymi ROBOT_POLECENIE i ROBOT_WARTOSC, które poprzez interfejs sprzętowy przekazywane są do oprogramowania niskopoziomowego. Po pomyślnym wykonaniu komendy, co zajmuje określony czas, oprogramowanie to przesyła do maszyny CPDev1 identyfikator ID_RFID i potwierdzenie ID_OK. Należy dodać, że w przypadku gdy KOMENDA wskazywała NA_WPROST, ID_RFID jest nowo zidentyfikowanym znacznikiem, natomiast w przypadku OBR_LEWO lub OBR_PRAWO, ID_RFID reprezentuje wcześniej zidentyfikowany znacznik w miejscu, w którym robot wykonał obrót. Nie jest on więc wpisywany do tablicy MAPA_RFID.

Jak wspominano wyżej, normalnym stanem flagi ID_OK jest FALSE oznaczający gotowość do odczytu. Jedynie po odczytaniu nowego identyfikatora ID_RFID robot ustawia flagę ID_OK na TRUE. Zmiany ID_OK wyznaczają cykle, w których wykonywane są zasadnicze fragmenty programów CPDEV1 i CPDEV3 (Dodatek B).

7.4. Wspólny projekt CPDev dla systemu eksploracji

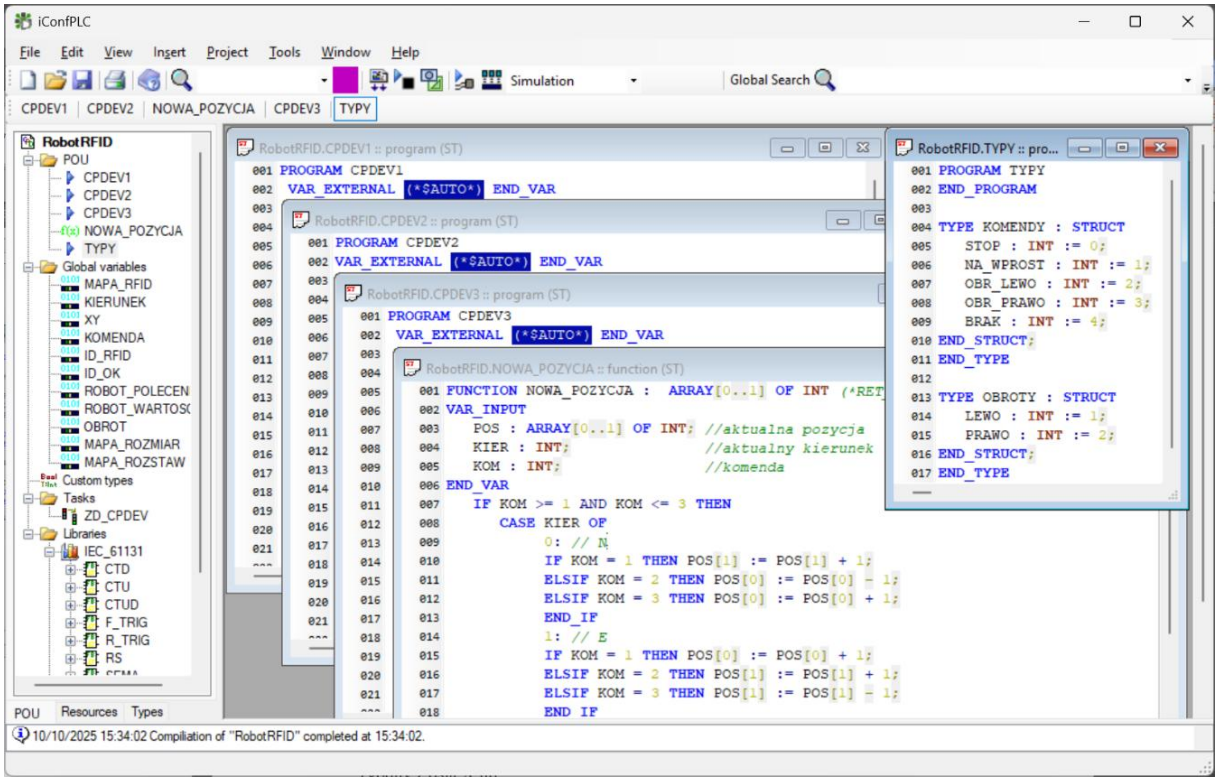
Oprogramowanie realizujące eksplorację opracowano najpierw jako jeden wspólny projekt RobotRFID, który po symulacjach został rozdzielony na jednostkę nadrzędną i robota mobilnego. Na projekt ten składają się programy CPDEV1, CPDEV2, CPDEV3 przeznaczone dla maszyn wirtualnych, funkcja NOWA_POZYCJA oraz specjalny program TYPY. Składniki te pokazano w zbiorczej formie na rysunku 7.6 rozpoczynając od programu TYPY. Kody źródłowe pozostałych składników wraz z wyjaśnieniami znajdują się w Dodatku B. Zmiennymi globalnymi są wszystkie zmienne wymienione w tabeli 7.1. Dyrektywa `VAR_EXTERNAL (*$AUTO*) END_VAR` importuje je do danego programu.

Program TYPY definiuje nowe typy strukturalne KOMENDY i OBROTY składające się z wartości INT. Wykorzystanie tych typów w innych programach wymaga utworzenia w nich zmiennych lokalnych o takich typach, czyli np. `KOM:KOMENDY` i `OBR:OBROTY`. Wówczas konkretne komendy mają postać `KOM.NA_WPROST`, `KOM.OBR_LEWO` itp.

Zmienna KIERUNEK jest kodowana jako 0,1,2,3, co umownie odpowiada orientacji N,E,S,W. Tak więc KIERUNEK 0 jako N jest zgodny z osią Y (wzrost), KIERUNEK 1 zgodny z osią X itd.

Projekt RobotRFID został wstępnie przetestowany jako zadanie ZD_CPDEV wykorzystując symulator wbudowany w środowisko IDE, który dla złożonych projektów okazuje się wygodniejszy niż CPSim. Tym razem pakiet CPDev wykorzystano w wersji iConfPLC przystosowanej dla hiszpańskiej firmy iGrid [26]. Celem testowania było

sprawdzenie, czy współpraca programów CPDEV1, CPDEV2 i CPDEV3 wygląda tak jak opisano w poprzednim punkcie. Testowanie polegało na wprowadzaniu danych powodujących generowanie KOMEND zależnych od położenia oraz na nadpisywaniu wartości zmiennych ID_RFID i ID_OK (które generowałby robot). Przykładowe wyniki są pokazane na rysunku 7.7.



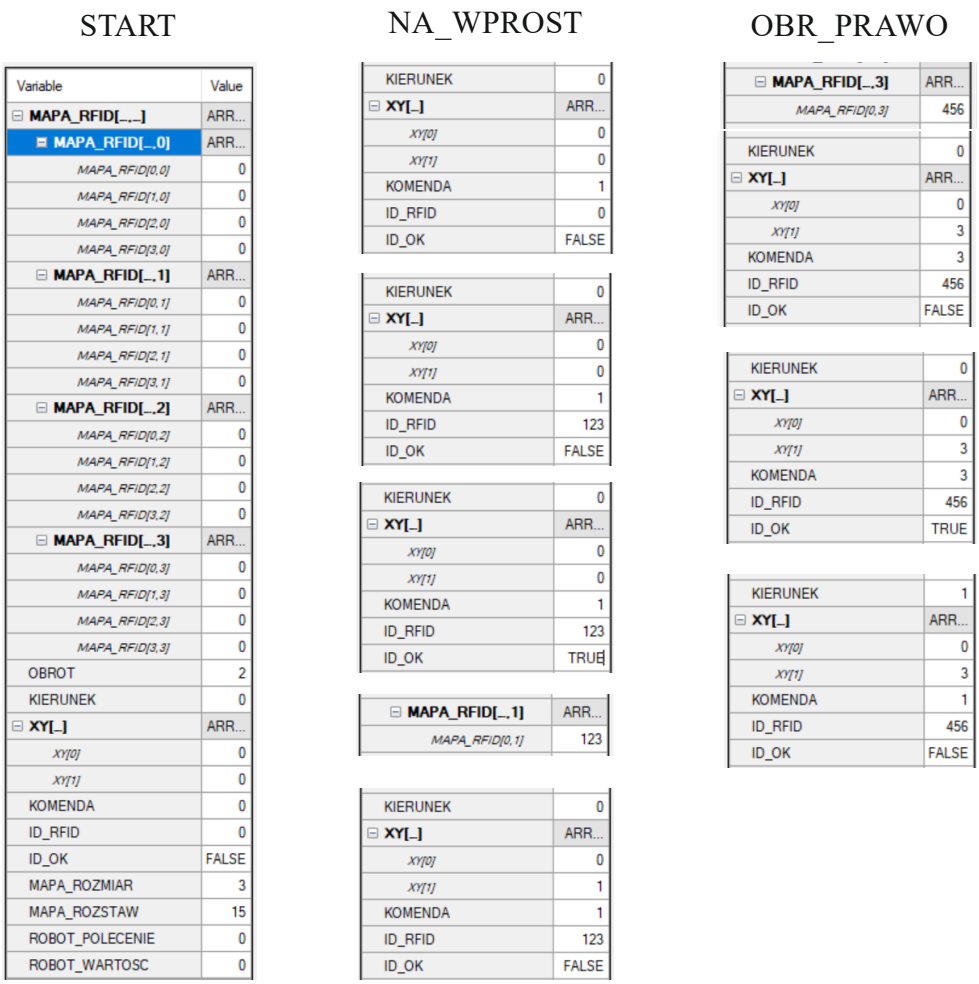
Rys. 7.6. Wspólny projekt RobotRFID.

Lewa kolumna rysunku reprezentuje stan początkowy, w którym tablica MAPA_RFID jest wyzerowana, KIERUNEK 0 oznacza oś Y, OBROT 2 narzuca obroty w prawo, a 3 jako MAPA_ROZMIAR określa tablicę 4×4. Druga kolumna zawiera kolejne wyniki symulujące zachowanie robota znajdującego się na pozycji [0,0], czyli w lewym dolnym rogu siatki. Jak pokazuje górny fragment kolumny, system wygenerował KOMENDĘ 1, czyli NA_WPROST, potem odczytał ID_RFID jako 123 (drugi fragment, nadpisane nową wartością), po czym ustawił ID_OK na TRUE (nadpisane). Identyfikator 123 jako element [0,1] został wpisany do tablicy MAPA_RFID. Dolny fragment informuje, że następną wartością zmiennej KOMENDA jest 1, czyli NA_WPROST, a flaga ID_OK jako FALSE (nadpisane) sygnalizuje gotowość do następnego odczytu.

Trzecia kolumna na rysunku 7.7 dotyczy sytuacji, gdy robot znajduje się w lewym górnym rogu, tzn. na pozycji XY [0,3], na której wcześniej posuwając się NA_WPROST odczytał identyfikator 456. Wygenerowaną komendą jest teraz 3 – OBR_PRAWO, po

wykonaniu której KIERUNEK z dotychczasowego 0 (oś Y) zmienia się na 1 (oś X). Pozycja XY nie ulega zmianie, bo obrót został wykonany w miejscu. Następną komendą jest 1 – NA_WPROST wzdłuż osi X.

Tego typu symulacje potwierdziły zdolność projektu RobotRFID do eksploracji według algorytmu z punktem końcowym w środku siatki (rys. 7.3).

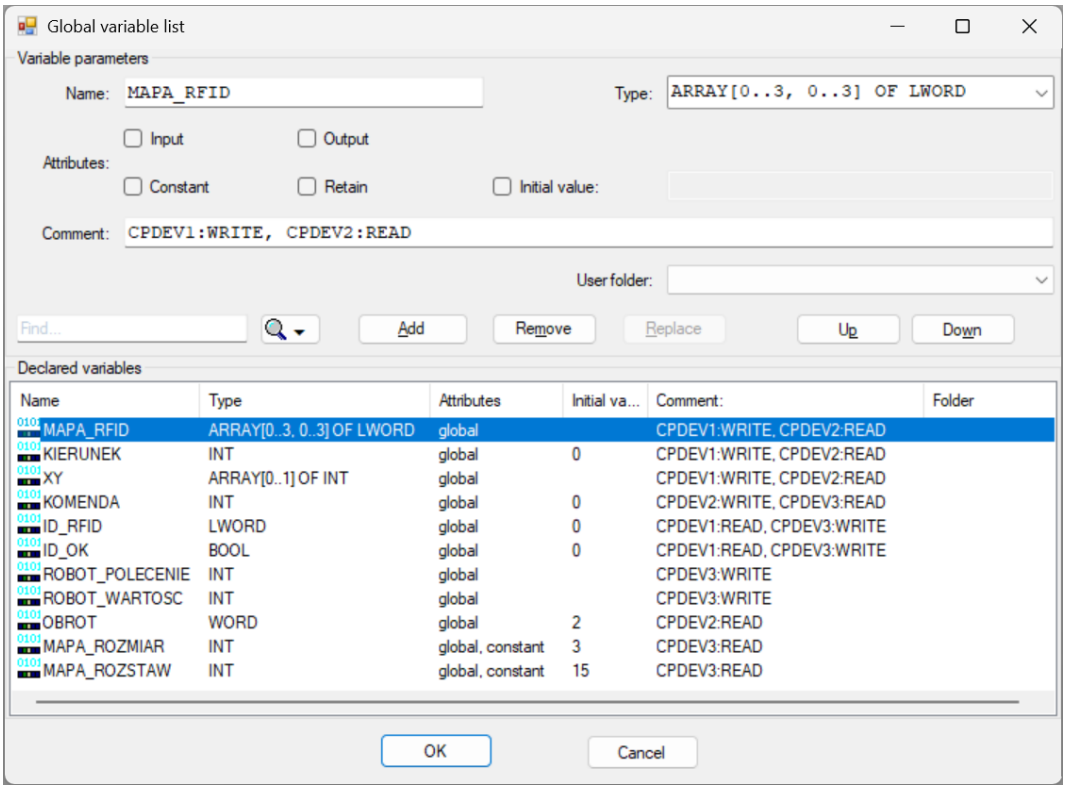


Rys. 7.7. Okna CPDev w trybie symulacji projektu RobotRFID.

7.5. Implementacja laboratoryjna

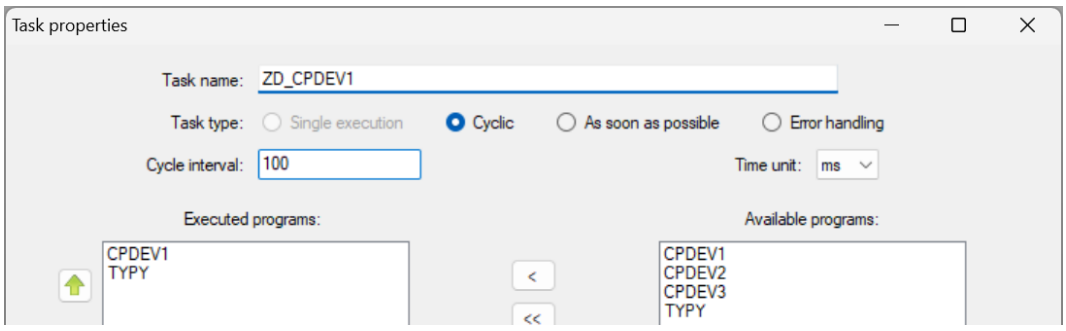
Podział wspólnego projektu. Wspólny projekt RobotRFID byłby odpowiedni dla implementacji jednordzeniowej, stąd atrybuty READ/WRITE definiujące wymianę międzyrdzeniową były w nim nieistotne. Są one jednak potrzebne przy podziale projektu na odrębne zadania przypisywane rdzeniom, zgodnie z logiką wymiany zmiennych podaną w tabeli 7.1. W związku z tym odnośne deklaracje w *Global variable list* mają postać jak na rysunku 7.8, z komentarzami *Comment* określającymi te atrybuty w odniesieniu do konkretnych zmiennych (por. rys. 6.2). Na przykład zmienna MAPA_RFID jest zapisywana – WRITE w pamięci współdzielonej przez rdzeń z programem CPDEV1, a odczytywana – READ przez

rdzeń z programem CPDEV2. Zmienną KOMENDA zapisuje CPDEV2, a odczytują CPDEV1 i CPDEV3, z tym że w pierwszym przypadku następuje to poprzez pamięć współdzieloną, a w drugim za pośrednictwem Wi-Fi.



Rys. 7.8. Deklaracje wymienianych zmiennych globalnych.

Podobnie jak przy podziale wspólnego projektu w poprzednim rozdziale (pkt. 6.2), dla każdego z rdzeni tworzone jest osobne zadanie, czyli tutaj ZD_CPDEV1, ZD_CPDEV2 i ZD_CPDEV3, na podstawie którego kompilator generuje osobny kod wykonywalny implementowany w danym rdzeniu. Każde z tych zadań składa się ze wskazanego programu CPDEV oraz programu TYPY definiującego typy strukturalne. Przykład tworzenia zadania ZD_CPDEV1 pokazano na rysunku 7.9.



Rys. 7.9. Programy tworzące zadanie ZD_CPDEV1.

Charakterystyka ogólna. Jak pokazano na rysunku 7.2, w skład laboratoryjnego systemu eksploracji RFID wchodzi jednostka nadrzędna, robot mobilny oraz płyta ze znacznikami. Zewnętrzny interfejs zapewnia komputer PC lub inne urządzenie z komunikacją Wi-Fi. Znaczniki rozmieszczone są w siatce kwadratowej o boku 15 cm. Średnica okręgu odczytu znacznika przez czytnik RFID robota wynosi około 4 cm.

System eksploracji jest systemem heterogenicznym ze względu na mikrokomputer Raspberry Pi w jednostce nadrzędnej i mikrokontroler ESP32 w robocie. Wymaga to odmiennego podejścia do zarządzania zasobami każdego z nich (rys. 7.4). Rozumie się przez to komunikację Wi-Fi, obsługę pamięci i plików, dostępność bibliotek, strukturę oprogramowania oraz metody debugowania. Oprogramowanie mikrokomputera i *firmware* mikrokontrolera wspólnie oparto o język C/C++ zgodnie z bazową implementacją maszyny wirtualnej CPDev.

Na obydwu platformach zastosowano jednolite podejście polegające na izolacji rdzeni z maszynami CPDev (*CPU affinity*), co zwiększa determinizm czasowy działania systemu. Praktyka taka jak stosowana w sterownikach przemysłowych Beckhoffa [84], potwierdzając przydatność w projektach wymagających wysokiej niezawodności.

Każda maszyna wirtualna CPDev ma przydzielony sektor pamięci na dane, tzn. zmienne globalne i lokalne, czyli na pamięć lokalną LM. W systemie wieloprojektowym istnieje również dodatkowa współdzielona przestrzeń pamięci SM, będąca częścią wspólną pamięci LM wszystkich maszyn wirtualnych systemu. Aktualizacja obszarów pamięci realizowana jest jako osobne w czasie operacje odczytu i zapisu, skorelowane z aktualnym stanem cyklu sterownika. Zgodnie z punktem 5.3 zastosowano tutaj wspomniany mechanizm *process image*, który wspomaga współpracę deterministycznego cyklu pracy sterownika z niedeterministyczną transmisją bezprzewodową (Wi-Fi), zwiększając odporność systemu na opóźnienia.

Jednostka nadrzędna. Platformą jednostki nadrzędnej jest mikrokomputer Raspberry Pi z systemem operacyjnym Linux. Podstawowe funkcjonalności obejmują cztery zadania pokazane na rysunku 7.4, z których dwa wykonują kody maszyn wirtualnych CPDev1, CPDev2 przypisanych odpowiednio do rdzeni 3,4, z wymianą zmiennych poprzez SM, a dwa następne w rdzeniach 1,2 odpowiadają za synchronizację zmiennych wymienianych poprzez Wi-Fi (protokół MQTT). W celu obsługi wielozadaniowości oraz pomiaru czasu zaimportowano standardowe biblioteki `thread` i `chrono`. Wymianę danych pomiędzy zadaniami zrealizowano z wykorzystaniem pamięci współdzielonej, nad którą kontrolę zapewniają muteksy z biblioteki `mutex`. Komunikację według protokołu MQTT oparto na implementacji Paho MQTT, do której wiadomości są przesyłane za pomocą kolejek z biblioteki `queue`.

Implementując oprogramowanie w systemie Linux zastosowano następujące rozwiązania:

- `isolcpus` – parametr w pliku `cmdline.txt` izolujący określone rdzenie od systemowego planowania i harmonogramowania zadań,
- `pthread_setaffinity_np()` – funkcja przypisująca zadania do konkretnych rdzeni CPU,
- `clock_nanosleep()` – funkcja zapewniająca cykliczne uruchamianie zadań.

W celu wyeliminowania zmienności podstawy czasu wynikającej z dynamicznej zmiany częstotliwości CPU, czyli DVFS (*Dynamic Voltage and Frequency Scaling*), ustawiono stałą częstotliwość za pomocą parametrów `arm_freq=1500` i `force_turbo=1`, również w pliku `cmdline.txt`. Skutkiem ubocznym takiej konfiguracji jest zwiększony pobór mocy.

Standardowe jądra systemu Linux nie spełniają kryteriów twardych systemów czasu rzeczywistego (*hard real-time systems*) [85]. Jednakże istnieją specjalistyczne odmiany pozwalające uzyskać pożądany determinizm czasowy. Przykładem może być RTLinux, w którym jądro Linux działa jako wątek o najniższym priorytecie (*idle thread*) pod kontrolą nadzorczego jądra czasu rzeczywistego RTCore. Innym rozwiązaniem jest dodatek PREEMPT-RT [86], który umożliwia przekształcenie klasycznego jądra w wariant *real-time* poprzez odpowiednią konfigurację i rekompilację. Warto dodać, że PREEMPT-RT został niedawno włączony trwale do jądra Linux.

Protokół MQTT. Do wymiany danych pomiędzy jednostkami systemu wykorzystano protokół MQTT (*Message Queuing Telemetry Transport*) [87], umożliwiający asynchroniczną transmisję danych o wybranej jakości QoS (*Quality of Service*) w środowiskach rozproszonych. Dane przesyłane są w strukturalnym formacie JSON umożliwiającym analizę przesyłanych komunikatów w czasie pracy. Dla zapewnienia spójności i synchronizacji danych pomiędzy węzłami każdy komunikat MQTT zawiera dodatkowo znacznik czasowy i numer bieżącego cyklu inkrementowany przy każdym przebiegu głównej pętli zadaniowej. Taka struktura pozwala odfiltrować opóźnione komunikaty oraz odwzorować stan systemu w czasie, co ułatwia wykrywanie przerw i niespójności transmisji. Przykładowy komunikat z maszyny CPDev2 aktualizujący zmienną KOMENDA pokazano na listingu 7.1.

Robot mobilny. Robot WaveRover pokazany na rysunku 7.2 stanowi zwartą, czterokołową konstrukcję o ramie wykonanej z aluminium, z płytą sterującą zawierającą mikrokontroler ESP32-S3 (rys. 7.10a). Robot wyposażony jest w bezpośredni napęd, zintegrowane sterowniki silników, 9-osiowy moduł IMU, czytnik kart pamięci, interfejs do skanera LIDAR oraz układy sterowania serwomechanizmami. Modularna konstrukcja sprzętowo-programowa umożliwia

elastyczne rozszerzanie funkcjonalności i implementację nowych algorytmów. Każde z kół połączone jest z silnikiem prądu stałego (DC) z przekładnią. Koła zostały sprzęgnięte elektrycznie w pary po każdej stronie robota (jeden kanał sterownika na dwa silniki). Tylne koła dodatkowo wyposażono w magnetyczne enkodery umieszczone bezpośrednio na wałach silników. Taka konfiguracja umożliwia dość precyzyjne przemieszczanie robota w linii prostej pod warunkiem odpowiedniej nawierzchni. Kinematyka pojazdu utrudnia jednak obrót wokół własnej osi oraz poruszanie się po torach zakrzywionych. Ruch obrotowy realizowany jest poprzez różnicowe sterowanie napędem, najefektywniej przy przeciwbieżnym obracaniu kół.


```
{
  "cycle": 793,
  "shared_var": {
    "KOMENDA": 0
  },
  "timestamp_us": 1755854690740900
}
```


Listing 7.1. Komunikat w formacie JSON ustawiający wartość 0 zmiennej KOMENDA.

Moduł IMU. Integruje on trójosiowy akcelerometr i żyroskop w układzie QMI8658C z magnetometrem AK09918. Komunikacja z modułem realizowana jest poprzez interfejs I^2C z częstotliwością 40 Hz. Na bazie otrzymanych danych obliczane są kąty Eulera – *pitch*, *roll* i *yaw*. W kontekście sterowania ruchem kluczowy jest kąt *yaw*, odpowiadający za estymowaną orientację robota w płaszczyźnie poziomej. Do obsługi czujników i obliczeń kątów wykorzystano bibliotekę producenta robota odpowiednio dostosowaną do pozostałych komponentów systemu. Podstawą jest algorytm AHRS (*Attitude and Heading Reference System*) [88], rozszerzony o filtr Alpha (wykładniczy) dla każdego z czujników.

Czytnik RFID. Czytnik PN532 firmy NXP Semiconductors (rys. 7.10b) został zintegrowany przy użyciu biblioteki Adafruit-PN532 poprzez dedykowany do tego celu drugi interfejs I^2C. Rozwiązanie to miało na celu odseparowanie czasochłonnej komunikacji z czytnikiem od cyklicznego odczytu IMU, niezbędnego do estymacji orientacji robota. Czytnik pracuje w trybie cyklicznego odczytu z maksymalnym czasem oczekiwania 100 ms (*timeout*) na pojawienie się identyfikatora. Odczytane dane są wyświetlane na wbudowanym ekranie OLED oraz przesyłane do systemu sterowania.

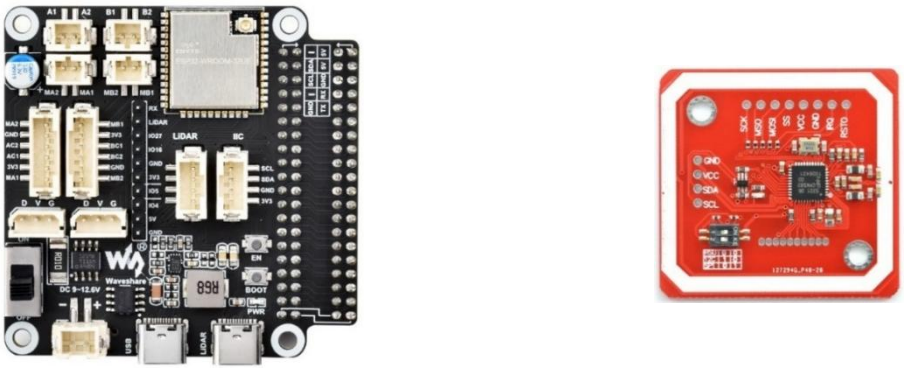
System wbudowany robota. Ma postać płyty na platformie ESP32 i wymaga zachowania determinizmu czasowego w obsłudze komunikacji bezprzewodowej (Wi-Fi). Dwa niezależne rdzenie ESP32, których zadania są sterowane przez wspólny system operacyjny czasu rzeczywistego FreeRTOS, umożliwiają logiczne odseparowanie maszyny wirtualnej CPDev3 przypisanej do rdzenia 2 od pozostałych zadań niezbędnych do poprawnego działania aplikacji.

W tym celu posłużono się funkcją systemową `xTaskCreatePinnedToCore`. Zadaniami implementowanymi w rdzeniu 1 są:

- sterowanie napędami według ROBOT_POLECENIE i ROBOT_WARTOSC,
- komunikacja z czytnikiem RFID,
- komunikacja MQTT z jednostką nadrzędną.

Bezpieczny dostęp do pamięci maszyny CPDev3 podczas komunikacji między zadaniami zapewniają muteksy systemu FreeRTOS.

a) b)



Rys. 7.10. a) Płyta sterująca z ESP32-S3, b) czytnik RFID PN532.

Sterowanie napędem. Oprogramowanie odpowiedzialne za sterowanie napędem zostało opracowane od podstaw rezygnując z nadmiernie rozbudowanego oprogramowania firmowego. Składają się na nie dwa podstawowe moduły funkcjonalne:

- `motorControl` – bezpośrednie sterowanie silnikami przez modulację stopnia wypełnienia impulsów PWM. Dla każdej ze stron robota impulsy z enkoderów wyzwalają przerwania, w których są zliczane, a następnie przeliczane na przebytą odległość na podstawie promienia koła.
- `motionControl` – realizacja przemieszczenia liniowego, obrotu lub ich kombinacji poprzez koordynację niezależnych zespołów napędowych na podstawie informacji zwrotnej z enkoderów i z modułu IMU. Dla ruchu liniowego zastosowano regulator PID, a dla obrotu algorytm typu *bang-bang*. Parametry obydwu regulatorów dobrano eksperymentalnie.

Dostrajanie. Podczas ruchu robota mobilnego występują rozmaite zakłócenia, związane w szczególności z chwilowymi poślizgami, błędami IMU, niesymetrią kół i różnicami momentów silników, czy brakiem zbieżności kół. W celu ograniczenia wpływu tych zakłóceń na przebieg eksploracji wprowadzono dodatkowe parametry poprawiające właściwości jazdy. Przykładowo, dla wyrównania napędów dodano regulator PID przenoszący część sygnału

sterowania na jedną ze stron na podstawie różnicy przebytego dystansu przez koła. Podobnie dla IMU zastosowano kalibrację w celu dostosowania kąta *yaw* do rzeczywistego kąta osiąganego przez robota.

Badanie eksploracji. W odróżnieniu od bezproblemowej symulacji projektu RFID przedstawionej w poprzednim punkcie, robot w systemie laboratoryjnym poddany jest zakłóceniom wymienionym wyżej. Dochodzą do tego niedokładności ustawień początkowych oraz ograniczona dokładność wykonywania komend. Czynniki te powodują, że nie będąc wyposażony w system automatycznej lokalizacji, robot stopniowo zbacza z trasy wyznaczonej przez algorytm i po pewnym czasie nie napotyka już na oczekiwany znacznik RFID. Jak podano w punkcie 7.1, robot ma się wtedy zatrzymać i sygnalizować potrzebę interwencji.

Sygnalizacja pominięcia oczekiwanego znacznika następuje po tym, gdy maszyna CPDev2 przesłała komendę do CPDev3 i wyprzedzająco do CPDev1. Od tego momentu maszyna CPDev1 oczekuje na odczyt. Interwencja polega na ręcznym ustawieniu robota nad ostatnio odczytanym znacznikiem na wprost kierunku ze znacznikiem pominiętym i poleceniu kontynuacji eksploracji. Przesuwający się znowu robot identyfikuje pominięty znacznik i jako ID_RFID z potwierdzeniem ID_OK przesyła go do CPDev1 (rys. 7.5). Z punktu widzenia maszyn wirtualnych pominięcie znacznika sprowadza się więc do dłuższego oczekiwania na jego odczyt ze względu na czas samej interwencji.

Wobec powyższego, faktycznym celem testów laboratoryjnych była ocena po identyfikacji ilu znaczników RFID w siatce 4×4 i planowanej trasie jak na rysunku 7.3 potrzebna będzie interwencja. Jak podano wcześniej, znaczniki były odległe od siebie o 15 cm, a średnica okręgu odczytu wynosiła około 4 cm.

Okazało się, że jedyna interwencja zewnętrzna jest potrzebna po identyfikacji 14 do 15 znaczników (5 obrotów), po której robot dociera już do środka siatki.[1]

Kluczową sprawą dla utrzymania robota na wytyczonej trasie jest dokładne wykonywanie obrotów o 90° pomimo odchyleń kinematycznych i poślizgów kół. Jak można było się spodziewać, identyfikatory umieszczone w tablicy MAPA_RFID były zawsze takie jak oczekiwano.

[1] Nagrania z przykładowych eksploracji: http://robotrfid.kia.prz.edu.pl/

8. Podsumowanie

W pracy przedstawiono metodę rozszerzenia maszyny wirtualnej, przeznaczonej oryginalnie dla środowiska jednordzeniowego na środowisko wielordzeniowe, w którym rdzenie procesora wykonują projekty sterowania normy IEC 61131-3 mogące wymieniać między sobą dane. Przez maszynę wirtualną należy rozumieć procesor zrealizowany programowo, wykonujący pewien uniwersalny kod pośredni wygenerowany przez kompilator programu źródłowego. Powszechnie stosowanymi maszynami wirtualnymi są JVM i CLR w środowiskach Java i .NET. Programowa realizacja maszyny pozwala implementować ją na różnych fizycznych procesorach, a do generacji uniwersalnego kodu pośredniego wystarcza jeden kompilator. Środowiskiem wykorzystującym maszynę wirtualną jest CPDev opracowany w PRz, który stanowił bazę dla niniejszej prący. CPDev jest zorientowany na sterowniki programowalne, co wyraża język pośredni VMASM, którego instrukcje odpowiadają bezpośrednio funkcjom normy IEC 61131-3 lub, jako procedury systemowe, dotyczą architektury maszyny. Na architekturę tę, podobnie jak w fizycznym procesorze, składają się pamięci kodu i danych, stosy, rejestry, moduł przetwarzania instrukcji oraz interfejs docelowej platformy sprzętowej. CPDev z tym samym kompilatorem i bazową maszyną wirtualną zastosowano w sterownikach przemysłowych z mikroprocesorami 16- i 32-bitowymi oraz w komputerach PC rodziny x86. Oryginalna maszyna wirtualna CPDev jest przeznaczona dla implementacji w jednym rdzeniu.

Jednakże od ponad dekady dostępne są już procesory wielordzeniowe, pozwalające rozdzielić program wykonywany przez jeden rdzeń na kilka rdzeni, co odpowiednio skraca czas wykonywania. Pojawiły się również pierwsze sterowniki z takimi procesorami. Z praktycznego punktu widzenia, reprezentatywny dla korzyści z dwurdzeniowości może być problem automatyzacji obiektu technologicznego wymagającego zarówno sterowania logicznego typu PLC, jak i powiązanej z nim regulacji PID. Dotychczas automatyzacja takiego obiektu wymagałaby zastosowania dwóch komunikujących się urządzeń, jednego PLC, drugiego PID, albo sterownika z systemem operacyjnym RTOS i zadaniami funkcjonującymi jako PLC i PID. Każde z tych rozwiązań ma jednak określone wady. W pracy pokazano jak można byłoby rozwiązać tego typu problem stosując procesor dwurdzeniowy, z rdzeniami jako PLC i PID, które wymieniają między sobą dane poprzez pamięć współdzieloną.

W wyniku analizy normy IEC 61131-3 oraz dwóch środowisk programistyczno-wykonawczych z systemami RTOS zaproponowano, aby na metodę rozszerzenia środowiska jednordzeniowego na dwa lub więcej rdzeni składały się następujące zasady:

1) jednakowe listy zmiennych globalnych wymienianych między rdzeniami deklarowane w projektach dla tych rdzeni,
2) dodatkowe atrybuty tych zmiennych w każdym projekcie określające tryb dostępu do pamięci współdzielonej,
3) organizacja wymiany zmiennych poprzez fazy *precycle* i *postcycle* cyklu każdego rdzenia,
4) eliminacja konfliktów rdzeni przy dostępie do pamięci współdzielonej.

Dwie pierwsze zasady dotyczą środowiska programistycznego, a dwie następne – wykonawczego. Do eliminacji konfliktów można wykorzystać semafor sprzętowy lub programowy, albo mechanizm wzajemnego wykluczania muteks. Warto zwrócić uwagę, że powyższe zasady nie wskazują, czy środowisko wykonawcze ma bazować na fizycznej jednostce CPU, czy na maszynie wirtualnej. Ponieważ dla dotychczasowego środowiska CPDev pamięć współdzielona jest nowym elementem architektury, więc rozszerzenie maszyny wirtualnej na więcej rdzeni wymagało uzupełnienia języka VMASM o nowe procedury systemowe dotyczące tej pamięci.

Ze względu na unikalną architekturę i odpowiadający jej język VMASM funkcjonowanie maszyny wirtualnej przedstawiono za pomocą semantycznych modeli denotacyjnych wyrażających znaczenie zastosowanych algorytmów. Z publikacji współpracowników zaczerpnięto model wyboru wykonywanej instrukcji ze skompilowanego programu na podstawie identyfikatora cyfrowego oraz przykładowe modele wykonywania instrukcji VMASM. Wkładem pracy są natomiast modele procedur obsługi pamięci współdzielonej z oczekiwaniem na bezkonfliktowy dostęp, wyrażony w postaci stałopunktowego równania denotacyjnego z pętlą *while*. Ponieważ modele denotacyjne wizualnie odpowiadają programom, więc można je implementować w powszechnie stosowanych językach, jak to pokazano dla kilku przykładów w C/C++. Jednak w odróżnieniu od czytelnych modeli denotacyjnych, zoptymalizowane implementacje C/C++ są dość trudne do analizy.

Ze względu na znaczenie praktyczne, jeden z wprowadzających rozdziałów dotyczył badania wpływu metod dostępu do pamięci na wydajność maszyny wirtualnej i rozmiar pamięci kodu. Porównywano dostęp bajtowy, kopiowanie *memcpy* oraz bezpośredni dostęp wskaźnikowy w odniesieniu do maszyny 16- i 32-bitowej oraz 32-bitowej z wyrównaniem adresacji do 4 bajtów. Wyniki badań pozwalają wnosić, które z metod dostępu można polecać dla konkretnej wersji maszyny biorąc również pod uwagę wymagania projektowe.

Pierwszy przykład zastosowania rozszerzonej maszyny wirtualnej dotyczył wykonywania dwóch współpracujących projektów IEC 61131-3 przez dwa rdzenie mikrokontrolera STM32 korzystając tylko z oprogramowania narzędziowego producenta (tryb *bare-metal*). Projekty napisane w językach FBD i ST mogły być od razu kompilowane osobno albo połączone najpierw we wspólny projekt i dopiero potem rozdzielone na dwa rdzenie. Symulację współpracy rdzeni, których programy wykonywane są z różnymi cyklami przeprowadzono za pomocą wielowątkowego WinControllera. Do eliminacji konfliktów rdzeni w dostępie do pamięci współdzielonej wykorzystano sprzętowy semafor mikrokontrolera STM32. Jak wspomniano wcześniej, do konwencjonalnej realizacji przykładu można byłoby wykorzystać jednordzeniowy sterownik z systemem RTOS lub dwa prostsze, ale komunikujące się sterowniki. Dwurdzeniowy sterownik typu *bare-metal* wyglądałby więc korzystniej.

Drugim przykładem był laboratoryjny system złożony z robota mobilnego z mikrokontrolerem ESP32 oraz mikrokomputera Raspberry Pi jako jednostki nadrzędnej, komunikującej się z robotem poprzez Wi-Fi. System identyfikował znaczniki RFID (transpondery) za pomocą czytnika w robocie. Na oprogramowanie składały się trzy maszyny wirtualne CPDev, dwie w Raspberry Pi i jedna w ESP32, przypisane do izolowanych rdzeni. Oprogramowanie utworzono najpierw jako jednordzeniowy wspólny projekt, który po przetestowaniu w IDE został rozdzielony na maszyny wirtualne. Wymiana danych poprzez Wi-Fi odbywała się według tych samych zasad, co przez pamięć współdzieloną. Systemy operacyjne Linux w Raspberry Pi i FreeRTOS w robocie zostały wyłączone z harmonogramowania zadań rdzeni, do których przypisano maszyny wirtualne. FreeRTOS zarządzał niskopoziomowym oprogramowaniem robota. W sumie zrealizowany system pokazał, że możliwe jest opracowanie złożonego systemu funkcjonującego w czasie rzeczywistym opartego na komunikujących się rozszerzonych maszynach wirtualnych.

Dodatek A. Podstawowe elementy modelu denotacyjnego

Model denotacyjny maszyny wirtualnej oprócz przedstawionych w punkcie 3.3 definicji stanu maszyny oraz uniwersalnej funkcji $\mathcal{U}$ wywołującej instrukcje języka VMASM, bazuje na dziedzinach semantycznych, funkcjach dziedzinowych oraz modelach instrukcji VMASM. Dziedziny, funkcje dziedzinowe oraz trzy modele instrukcji VMASM zaczerpnięte z [54,55] i poszerzone o implementacje w C/C++ są podane poniżej. Materiały źródłowe na temat denotacji i semantyki języków programowania można znaleźć w [32,33].

A.1. Dziedziny semantyczne

Dziedziny te reprezentują abstrakcyjne typy danych modelujące wartości przetwarzane przez maszynę wirtualną. W szczególności, dziedzina

$$BasicTypes = OneByte \times TwoBytes \times FourBytes \times EightBytes \tag{A.1}$$

jest złożeniem czterech bajtowych zbiorów reprezentujących rozmiary podstawowych typów danych IEC 61131-3. Dziedzina

$$Address = \boldsymbol{if}\ AddressSize = 2\ \boldsymbol{then}\ TwoBytes\ \boldsymbol{else}\ FourBytes \tag{A.2}$$

określa implementację z adresacją dwubajtową lub czterobajtową. Dziedziny podane niżej odpowiadają wprost architekturze maszyny wirtualnej (pkt. 3.1).

$$\begin{aligned} Memory &= Address \rightarrow OneByte \\ CodeMemory &= Memory \\ DataMemory &= Memory \\ Stack &= Address* \\ CodeStack &= Stack \\ DataStack &= Stack \\ CodeReg &= Address \\ DataReg &= Address \\ Flags &= TwoBytes \end{aligned} \tag{A.3}$$

Dziedzina $Memory$ jest funkcją odwzorowującą $Address$ na $OneByte$, a jej aliasami są $CodeMemory$ i $DataMemory$. $Stack$ reprezentuje sekwencję obiektów (oznaczoną symbolem $*$) z dziedziny $Address$ z aliasami $CodeStack$ i $DataStack$. Dziedziny $CodeReg$ i $DataReg$ reprezentują rejestry bazowe adresów dla kodu i danych. Dwubajtowy rejestr $Flags$ przechowuje flagi statusowe. Powyższe dziedziny składają się na zdefiniowaną w punkcie 3.3 dziedzinę $State$ określającą stan maszyny.

A.2. Funkcje dziedzinowe

Podane niżej funkcje modelują niskopoziomowe operacje między dziedzinami adresów, pamięci, typów danych, stosów, rejestrów i flag.

- Pobieranie z pamięci (odczyt) danych o określonym rozmiarze

$$\begin{aligned} Get1BMem &= (Address \times Memory) \rightarrow OneByte \\ Get2BMem &= (Address \times Memory) \rightarrow TwoBytes \\ Get4BMem &= (Address \times Memory) \rightarrow FourBytes \\ Get8BMem &= (Address \times Memory) \rightarrow EightBytes \end{aligned} \tag{A.4}$$

- Pobieranie adresu z pamięci

$$GetAddress = (Address \times Memory) \rightarrow Address \tag{A.5}$$

Funkcja zwraca wartość przechowywaną pod danym adresem w pamięci, który jest innym adresem (adresacja pośrednia). Jak podawano w rozdziale 3, maszyna wirtualna nie ma rejestru akumulatora i działa bezpośrednio na adresach, stąd funkcja $GetAdddress$ jest szczególnie istotna dla modelu.

- Aktualizacja pamięci (zapis) danych o określonym rozmiarze

$$\begin{aligned} Upd1BMem &= (Address \times Memory \times OneByte) \rightarrow Memory \\ Upd2BMem &= (Address \times Memory \times TwoBytes) \rightarrow Memory \\ Upd4BMem &= (Address \times Memory \times FourBytes) \rightarrow Memory \\ Upd8BMem &= (Address \times Memory \times EightBytes) \rightarrow Memory \end{aligned} \tag{A.6}$$

- Przenoszenie pamięci (kopiowanie)

$$MemMove = \begin{pmatrix} Address \times Memory \times Address \\ \times Memory \times OneByte \end{pmatrix} \rightarrow Memory \tag{A.7}$$

gdzie pierwszy $Address$ reprezentuje źródło, drugi cel, a $OneByte$ liczbę kopiowanych bajtów (ograniczoną do 255).

- Funkcje stosu

$$\begin{aligned} Push &= (Stack \times Address) \rightarrow Stack \\ Pop &= Stack \rightarrow (Address \times Stack) \end{aligned} \tag{A.8}$$

Funkcje wykonują operacje potrzebne w podprogramach. Warto zauważyć, że Pop zwraca parę, tj. adres i nową zawartość stosu.

- Operacje na flagach

$$ClearFlag = (TwoBytes \times TwoBytes) \rightarrow TwoBytes$$
$$SetFlag = (TwoBytes \times TwoBytes) \rightarrow TwoBytes \quad \text{(A.9)}$$

Kolejne $TwoBytes$ oznaczają aktualne flagi, bity resetowane lub ustawiane oraz nowe flagi.

- Operacje arytmetyczne w ograniczonym zakresie

Uniknięcie nadmiaru (*overflow*) w operacjach arytmetycznych wymaga rozszerzenia definicji takich operacji z ograniczaniem wyniku do dopuszczalnego zakresu. W przypadku dodawania liczb całkowitych ze znakiem dodawanie w ograniczonym zakresie jest zdefiniowane następująco

$$a \oplus b = \textbf{\textit{if}}\ (a+b) > MaxRange(a)\ \textbf{\textit{or}}\ (a+b) < MinRange(a)$$
$$\textbf{\textit{then}} - MinRange(a) + (a+b)\ mod\ \big(-MinRange(a)\big)$$
$$\textbf{\textit{else}}\ a+b \quad \text{(A.10)}$$

gdzie np. $MaxRange$ dla INT, DINT, LINT wynosi odpowiednio 32767, 2^{31}-1, 2^{63}-1. Podobne zdefiniowane jest mnożenie $\otimes$ i dzielenie $\oslash$.

- Pola bajtowe jako dane IEC 61131-3

Powiązanie bajtowych pól pamięci z typami danych IEC 61131-3 realizują następujące funkcje interpretujące

$$\begin{array}{ll} FromBool = BOOL \rightarrow OneByte & BoolOf = OneByte \rightarrow BOOL \\ FromByte = BYTE \rightarrow OneByte & ByteOf = OneByte \rightarrow BYTE \\ FromInt = INT \rightarrow TwoBytes & IntOf = TwoBytes \rightarrow INT \\ FromDInt = DINT \rightarrow FourBytes & DIntOf = FourBytes \rightarrow DINT \\ FromLInt = LINT \rightarrow EightBytes & LIntOf = EightBytes \rightarrow LINT \\ FromReal = REAL \rightarrow FourBytes & RealOf = FourBytes \rightarrow REAL \end{array} \quad \text{(A.11)}$$

Wymienionym powyżej funkcjom dziedzinowym odpowiadają niskopoziomowe funkcje C/C++ w maszynie wirtualnej.

A.3. Przykłady modeli VMASM i ich implementacje w C/C++

Funkcja `NOT` z tabeli A.1a poniżej neguje wartość operandu $op1$ umieszczając rezultat w r. Po rozłożeniu stanu s na krotkę, najpierw wyznaczone są adresy $raddr$, $op1addr$ z uwzględnieniem zawartości rejestru danych dr, wraz z inkrementacją rejestru kodu do cr_2 wskazującą na następną instrukcję (por. `ADD` w tab. 3.1). Następnie, poprzez odczyt bajtu funkcją $Get1BMem$ z pamięci danych dm i konwersję $BoolOf$ tworzona jest zmienna

boolowska bv reprezentująca $op1$. Końcowa funkcja $Upd1BMem$ aktualizuje rezultat pod adresem $raddr$ po konwersji $FromBool$ zanegowanego bv poprzez ***match***. W nowym stanie maszyny s_1 występuje nowy stan pamięci danych um oraz nowa zawartość cr_2 rejestru kodu. Implementacja C/C++ funkcji NOT po prawej stronie tabeli odpowiada bezpośrednio denotacjom przy założeniu, że `GetCodeAddress` inkrementuje również rejestr kodu.

Tab. A.1. Denotacje i implementacje w języku C/C++: a) funkcja NOT, b) funkcja EQ, c) procedura JNZ.

a)

Denotacja	**Implementacja w C**

$\mathcal{C}[\![\texttt{NOT:r:op1}]\!] = \lambda s.$
$(cm, dm, cs, ds, cr, dr, flg) := s$
$r := GetAddress(cr, cm)$
$raddr := dr \oplus r$
$cr_1 := cr \oplus AddressSize$
$op1 := GetAddress(cr_1, cm)$
$op1addr := dr \oplus op1$
$cr_2 := cr_1 \oplus AddressSize$
$bv :=$
$\quad BoolOf(Get1BMem(op1addr, dm))$
$um :=$
$\quad Upd1BMem(raddr, dm, FromBool($
$\quad\quad \textbf{match}\ bv\ \textbf{with}$
$\quad\quad\quad \mid \textbf{true} \rightarrow \textbf{false}$
$\quad\quad\quad \mid \textbf{false} \rightarrow \textbf{true}$
$\quad\quad \textbf{end}))$
$s_1 := (cm, um, cs, ds, cr_2, dr, flg)$
s_1
	`ADDRESS raddr =`
	`  dataReg + GetCodeAddress();`
	`ADDRESS op1addr =`
	`  dataReg + GetCodeAddress();`
	`BOOL bv =`
	`  BOOLOf(Get1BMemData(op1addr));`
	`Upd1BMemData(raddr,FromBOOL`
	`  (bv?FALSE:TRUE));`

b)

Denotacja	**Implementacja w C**

$\mathcal{C}[\![\texttt{EQ:LINT:r:op1:op2}]\!] = \lambda s.$
$(cm, dm, cs, ds, cr, dr, flg) := s$
$r := GetAddress(cr, cm)$
$raddr := dr \oplus r$
$cr_1 := cr \oplus AddressSize$
$op1 := GetAddress(cr_1, cm)$
$op1addr := dr \oplus op1$
$cr_2 := cr1 \oplus AddressSize$
$op2 := GetAddress(cr_2, cm)$
$op2addr := dr \oplus op2$
	`#define EQ_TYPE(TYPE)`
	`case IT_EQ_##TYPE & 0x000F: {`
	`  ADDRESS raddr = dataReg +`
	`  GetCodeAddress();`
	`  ADDRESS op1addr = dataReg +`
	`  GetCodeAddress();`
	`  ADDRESS op2addr = dataReg +`
	`  GetCodeAddress();`
	`  TYPE op1 =`
	`  TYPE##Of(GetMemData(op1addr,`
	`  sizeof(TYPE)));`

$cr_3 := cr_2 \oplus AddressSize$
$cmp :=$
$LIntOf(Get8BMem(op1addr, dm))$
$= LIntOf(Get8BMem(op2addr, dm))$
$s_1 := (cm, Upd1BMem(raddr, dm,$
$FromBool(cmp)), cs, ds, cr_3, dr, flg)$
s_1
	TYPE op2 =
	TYPE##Of(GetMemData(op2addr,
	sizeof(TYPE)));
	BOOL cmp = op1 == op2;
	Upd1BMemData(raddr,
	FromBOOL(cmp));
	} break;
	void IG_EQ_12(BYTE it) {
	switch (it & 0x0F){
	EQ_TYPE(SINT);
	EQ_TYPE(INT);
	EQ_TYPE(DINT);
	EQ_TYPE(LINT);
	... /* other types */
	default: /* unknown code */
	flag |= FAULT;}
	return; }

c)

Denotacja	**Implementacja w C**

$\mathcal{C}[\![\texttt{JNZ:cnd:clb1}]\!] = \lambda s.$
$(cm, dm, cs, ds, cr, dr, flg) := s$
$cnd := GetAddress(cr, cm)$
$cndaddr := dr \oplus cnd$
$cr_1 := cr \oplus AddressSize$
$clbl := GetAddress(cr_1, cm)$
$cr_2 := cr_1 \oplus AddressSize$
$ctl :=$
$BoolOf(Get1BMem(cndaddr, dm))$
$s_1 := (\textbf{match}\ ctl\ \textbf{with}$
$\mid \textbf{true} \rightarrow$
$(cm, dm, cs, ds, clbl, dr, flg)$
$\mid \textbf{false} \rightarrow$
$(cm, dm, cs, ds, cr_2, dr, flg)$
$\textbf{end})$
s_1
	void IG_SYSPROC_1C(BYTE it){
	switch(it){
	case 0x01:
	ADDRESS cndaddr = dataReg
	+ GetCodeAddress();
	ADDRESS clbl=
	GetCodeAddress();
	BOOL ctl = BOOLOf(
	Get1BMem(cndaddr));
	if(ctl)
	codeReg=clbl;
	break;
	…
	default:
	flags |= FAULT;
	break;}}

Funkcja `EQ` w tabeli A.1b sprawdza równość dwóch operandów typu LINT. Adresy określane są jak poprzednio z rejestrem kodu inkrementowanym do cr_3. Wartość logiczna cmp wynika z porównania (=) tych liczb utworzonych przez $LIntOf$ z odczytywanych ośmiu bajtów. Zaktualizowana pamięć danych w nowym stanie s_1 jest wynikiem $Upd1BMem$, przy czym bajt umieszczony w $raddr$ pochodzi z $FromBool(cmp)$. Podobnie jak w `ADD` z tabeli 3.1, funkcja `IG_EQ_12` po prawej stronie na dole implementuje porównanie wszystkich odpowiednich typów danych (przeciążenie). Wywołuje ona sparametryzowane makrodefinicje

`EQ_TYPE`, wspólne dla wszystkich typów. Funkcja `IG_EQ_12` rozpoznaje określony typ jako drugi półbajt identyfikatora `it` maskując go wartością `0x0F`.

Procedura `JNZ` w tabeli A.1c wykonuje skok do etykiety $clbl$ w pamięci kodu w zależności od warunku cnd w pamięci danych. Po określeniu adresu $cndaddr$ inkrementacja rejestru kodu do cr_1 wskazuje na następny operand, czyli adres, gdzie znajduje się etykieta $clbl$. Odczytuje się ją poprzez $GetAddress$. Wartość zmiennej boolowskiej ctl odczytywana z $cndaddr$ warunkuje wykonanie skoku. Jeżeli warunek jest spełniony, to nowy stan s_1 zawiera $clbl$, a nie cr_2 w rejestrze kodu. Realizacja C/C++ podana w tabeli odpowiada wprost denotacjom, z tym, że poprzedza ją wybór właściwiej procedury poprzez `switch(it)`. Model denotacyjny jeszcze jednej procedury, tj. MCD inicjującej dane w pamięci, jest podany w punkcie 4.3.

Dodatek B. Kody źródłowe projektu RobotRFID

Przedstawiono kody ST i komentarze dotyczące funkcji NOWA_POZYCJA oraz programów CPDEV1, CPDEV2, CPDEV3 wchodzących w skład projektu RobotRFID z punktu 7.4.

NOWA_POZYCJA. Funkcja pokazana na rysunku B.1 generuje przewidywane współrzędne robota w odniesieniu do aktualnych POS i kierunku KIER w sytuacji gdyby wykonał on komendę KOM. Kody 0,1,2,3 dla KIER odpowiadają orientacji N,E,S,W, np. N – 0 jest zgodne z osią Y, a 1 – E zgodne z osią X. Kody komend KOM podano w programie TYPY (rys. 7.6). W przypadku komend OBR_LEWO i OBR_PRAWO funkcja zwraca współrzędne następnego pola po wykonaniu obrotu. Stąd na przykład dla KIER 1 – oś X, komenda KOM 2 – OBR_LEWO obracałaby robot na oś Y, na której następne pole w stosunku do aktualnego POS(1) na współrzędną POS(1)+1, jak w drugiej sekcji instrukcji CASE. Współrzędna X tego pola, czyli POS(0) nie ulega wtedy zmianie.

RobotRFID.NOWA_POZYCJA :: function (ST)

```
FUNCTION NOWA_POZYCJA :  ARRAY[0..1] OF INT (*RET_TYPE*)
VAR_INPUT
    POS : ARRAY[0..1] OF INT; //aktualna pozycja
    KIER : INT;               //aktualny kierunek
    KOM : INT;                //komenda
END_VAR
    IF KOM >= 1 AND KOM <= 3 THEN
       CASE KIER OF
            0: // N
            IF KOM = 1 THEN POS[1] := POS[1] + 1;
            ELSIF KOM = 2 THEN POS[0] := POS[0] - 1;
            ELSIF KOM = 3 THEN POS[0] := POS[0] + 1;
            END_IF
            1: // E
            IF KOM = 1 THEN POS[0] := POS[0] + 1;
            ELSIF KOM = 2 THEN POS[1] := POS[1] + 1;
            ELSIF KOM = 3 THEN POS[1] := POS[1] - 1;
            END_IF
            2: //S
            IF KOM = 1 THEN POS[1] := POS[1] - 1;
            ELSIF KOM = 2 THEN POS[0] := POS[0] + 1;
            ELSIF KOM = 3 THEN POS[0] := POS[0] - 1;
            END_IF
            3: // W
            IF KOM = 1 THEN POS[0] := POS[0] - 1;
            ELSIF KOM = 2 THEN POS[1] := POS[1] - 1;
            ELSIF KOM = 3 THEN POS[1] := POS[1] + 1;
            END_IF
         END_CASE
      END_IF
    NOWA_POZYCJA := POS;  (*RET_VALUE*)
END_FUNCTION
```

RobotRFID.CPDEV1 :: program (ST)

```
PROGRAM CPDEV1
 VAR_EXTERNAL (*$AUTO*) END_VAR
 VAR
  ID_OK_TRIG : R_TRIG;
  TMP : ARRAY[0..1] OF INT;
  KOM : KOMENDY;
  X : INT := 0;
  Y : INT := 1;
 END_VAR

 ID_OK_TRIG(CLK:=ID_OK);

 IF ID_OK_TRIG.Q THEN
   IF KOMENDA = KOM.OBR_LEWO THEN
      KIERUNEK := (KIERUNEK + 3) MOD 4;
   ELSIF KOMENDA = KOM.OBR_PRAWO THEN
      KIERUNEK := (KIERUNEK + 1) MOD 4;
   ELSIF KOMENDA = KOM.NA_WPROST THEN
      TMP := NOWA_POZYCJA(XY, KIERUNEK,
                          KOM.NA_WPROST);
      XY[X] := TMP[X];
      XY[Y] := TMP[Y];
      MAPA_RFID[XY[0],XY[1]] := ID_RFID;
   ELSIF KOMENDA = KOM.STOP THEN
      MAPA_RFID[XY[0],XY[1]] := ID_RFID;
   END_IF
  END_IF
END_PROGRAM
```

Rys. B.1. Kody ST funkcji NOWA_POZYCJA i programu CPDEV1.

CPDEV1. Zasadnicze instrukcje programu (rys. B.1) wykonywane są tylko raz po wykryciu przez przerzutnik R_TRIG narastającego zbocza flagi ID_OK, co świadczy o wykonaniu ostatniej KOMENDY. W przypadku komendy NA_WPROST, instrukcje te aktualizują MAPĘ_RFID odczytanym identyfikatorem ID_RFID oraz jego współrzędne XY, bądź aktualizują tylko KIERUNEK, gdy komendą było OBR_LEWO lub OBR_PRAWO. Pierwszym argumentem funkcji NOWA_POZYCJA są współrzędne XY przed wykonaniem

NA_WPROST. Do aktualizacji KIERUNKU po wykonaniu obrotu (w miejscu) zastosowano arytmetykę modulo.

CPDEV2. Program (rys. B.2) pełni w systemie rolę decyzyjną. Jeżeli robot znajduje się na zidentyfikowanym polu, o czym świadczy niezerowa wartość w tablicy MAPA_RFID na pozycji XY, podejmowana jest próba określenia nowej KOMENDY dla kontynuacji eksploracji. Najpierw badana jest możliwość ruchu NA_WPROST poprzez sprawdzenie, czy przewidywana pozycja TMP wyznaczona funkcją NOWA_POZYCJA spełnia ograniczenie MAPA_ROZMIAR i wskazuje na niezidentyfikowane PUSTE pole. W przypadku, gdy wymagania te nie są spełnione, badane są warianty OBR_LEWO i OBR_PRAWO zależnie od parametru OBROT, z analogicznym sprawdzeniem poprawności pozycji TMP i dostępności pola PUSTE. Możliwość wykonania jednego z tych trzech wariantów określa nową KOMENDĘ dla programu CPDEV3. Program CPDEV2 jest wykonywany wielokrotnie na tych samych danych.

RobotRFID.CPDEV2 :: program (ST)

```
PROGRAM CPDEV2
VAR_EXTERNAL (*$AUTO*) END_VAR
VAR
   PUSTE: LWORD := 0;
   X : INT := 0;
   Y : INT := 1;
   KOM : KOMENDY;
   OBR : OBROTY;
   TMP: ARRAY[0..1] OF INT;
END_VAR

IF MAPA_RFID[XY[X], XY[Y]] <> PUSTE THEN
   KOMENDA := KOM.STOP;

   TMP := NOWA_POZYCJA(XY, KIERUNEK, KOM.NA_WPROST);

   IF (TMP[X] >= 0 AND TMP[X] <= MAPA_ROZMIAR)
            AND (TMP[Y] >= 0 AND TMP[Y] <= MAPA_ROZMIAR) THEN
      IF MAPA_RFID[TMP[X], TMP[Y]] = PUSTE THEN
         KOMENDA := KOM.NA_WPROST; // Na wprost, kierunek bez zmian
      END_IF
   END_IF

   IF KOMENDA = KOM.STOP THEN
      IF OBROT = OBR.LEWO THEN
           TMP := NOWA_POZYCJA(XY, KIERUNEK, KOM.OBR_LEWO);

           IF (TMP[X] >= 0 AND TMP[X] <= MAPA_ROZMIAR)
                    AND (TMP[Y] >= 0 AND TMP[X] <= MAPA_ROZMIAR) THEN
              IF MAPA_RFID[TMP[X], TMP[Y]] = PUSTE THEN
                 KOMENDA := KOM.OBR_LEWO; // Obrót w lewo w miejscu
              END_IF
           END_IF
      ELSE
           TMP := NOWA_POZYCJA(XY, KIERUNEK, KOM.OBR_PRAWO);

           IF (TMP[X] >= 0 AND TMP[X] <= MAPA_ROZMIAR)
                    AND (TMP[Y] >= 0 AND TMP[X] <= MAPA_ROZMIAR) THEN
              IF MAPA_RFID[TMP[X], TMP[Y]] = PUSTE THEN
                 KOMENDA := KOM.OBR_PRAWO; // Obrót w prawo w miejscu
              END_IF
           END_IF
      END_IF
   END_IF
END_IF
END_PROGRAM
```

RobotRFID.CPDEV3 :: program (ST)

```
PROGRAM CPDEV3
 VAR_EXTERNAL (*$AUTO*) END_VAR
 VAR
    KOM : KOMENDY;
    ID_OK_TRIG : F_TRIG;
 END_VAR

 ID_OK_TRIG(CLK:=ID_OK);

 IF ID_OK_TRIG.Q THEN
    IF KOMENDA = KOM.NA_WPROST THEN
       ROBOT_POLECENIE := 1;
       ROBOT_WARTOSC := MAPA_ROZSTAW;
    ELSIF KOMENDA = KOM.OBR_LEWO THEN
       ROBOT_POLECENIE := 2;
       ROBOT_WARTOSC := 90;
    ELSIF KOMENDA = KOM.OBR_PRAWO THEN
       ROBOT_POLECENIE := 3;
       ROBOT_WARTOSC := -90;
    ELSE
       ROBOT_POLECENIE := 0;
    END_IF
 END_IF
END_PROGRAM
```

Rys. B.2. Kody ST programów CPDEV2, CPDEV3.

CPDEV3. Instrukcje programu (rys. B.2) wykonywane są tylko raz, ale po wykryciu przez przerzutnik F_TRIG opadającego zbocza flagi ID_OK sygnalizującego gotowość do

wykonania KOMENDY wygenerowanej przez CPDEV2. Odpowiednio do niej zmiennym ROBOT_POLECENIE i ROBOT_WARTOSC nadawane są odpowiednie wartości, które poprzez interfejs sprzętowy przekazywane są do niskopoziomowego oprogramowania sterującego robotem. Program CPDEV3 pełni więc rolę translatora otrzymanej KOMENDY na fizyczne rozkazy ruchu. W trakcie wykonania ruchu robot wykrywa identyfikator ID_RFID po czym ustawia flagę ID_OK na TRUE aktywując CPDEV1. Po upływie pewnego czasu przeznaczonego na dokończenie ruchu robot ustawia ID_OK z powrotem na FALSE sygnalizując gotowość do wykonania następnej KOMENDY.

Spis rysunków

Spis tabel i listingów

Wykaz skrótów

ARM	Advanced RISC Machine
AVR	Alf and Vegard's RISC, rodzina mikrokontrolerów
CLR	Common Language Runtime
CPCtrl	Control Program Controller
CPDev	Control Program Developer
CPSim	Control Program Simulator
CPU	Central Processing Unit
DCS	Distributed Control System
DM_TO_SM	Data Memory to Shared Memory
DMA	Direct Memory Access
DVFS	Dynamic Voltage and Frequency Scaling
FBD	Function Block Diagram
GNSS RTK	Global Navigation Satellite System Real Time Kinematic
GPIO	General Purpose Input/Output
GVL	Global Variable List
HAL	Hardware Abstraction Layer
HMI	Human-Machine Interface
HSEM	Hardware Semaphore
I/O	Input/Output
I^2C	Inter-Integrated Circuit (IIC)
IDE	Integrated Development Environment
IL	Instruction List
IMU	Inertial Measurement Unit
IPC	Industrial PC
JVM	Java Virtual Machine
LCF	Library Configuration File
LD	Ladder Diagram
MMU	Memory Management Unit
MQTT	Message Queuing Telemetry Transport

.NET	Microsoft .NET Framework/Platform
OLED	Organic Light-Emitting Diode
OPC	Open Platform Communications
PAC	Programmable Automation Controller
PID	Proportional-Integral-Derivative
PLC	Programmable Logic Controller
PLL	Phase-Locked Loop
POU	Program Organization Unit
PREEMPT-RT	Preemptible Real-Time Patch
RFID	Radio Frequency Identification
RTLinux	Real-Time Linux
RTOS	Real-Time Operating System
SCADA	Supervisory Control and Data Acquisition
SFC	Sequential Function Chart
SM_TO_DM	Shared Memory to Data Memory
SRAM	Static Random-Access Memory
ST	Structured Text
TCP	Transmission Control Protocol
TP	Time Pulse
VM	Virtual Machine
VM16	Virtual Machine 16-bit
VM32	Virtual Machine 32-bit
VM32A	Virtual Machine 32-bit with 4-byte Alignment
VMASM	Virtual Machine Assembler
VMP	Virtual Machine Platform

Wykaz oznaczeń

cm	*code memory,* pamięć kodu
dm	*data memory,* pamięć danych
sm	*shared memory,* pamięć współdzielona
cs	*code stack,* stos kodu
ds	*data stack,* stos danych
cr	*code register,* rejestr kodu
dr	*data register,* rejestr danych
flg	*flag,* flaga
$\mathcal{U}$	*universal semantic function,* uniwersalna funkcja semantyczna
$\mathcal{C}$	*specific semantic function,* specyficzna funkcja semantyczna

Bibliografia

[1] EN 61131-3:2013: *Programmable controllers – Part 3: Programming languages*. International Electrotechnical Commission, 2013.

[2] Kasprzyk J., *Programowanie sterowników przemysłowych*. WNT, 2014.

[3] Siemens: SIMATIC STEP 7 (TIA Portal). Dostęp online: https://www.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal/software/step7-tia-portal.html. [09.09.2025].

[4] Beckhoff Automation: TwinCAT 3. Dostęp online: https://www.beckhoff.com/pl-pl/products/automation/twincat/texxxx-twincat-3-engineering/. [09.09.2025].

[5] ABB: Freelance Engineering. Dostęp online: https://new.abb.com/control-systems/pl/essential-automation/freelance. [09.09.2025].

[6] Rzońca D., Sadolewski J., Stec A., Świder Z., Trybus B., Trybus L., Developing a multiplatform control environment. *Journal of Automation, Mobile Robotics & Intelligent Systems*, t. 13, nr 4, s. 73–84, 2019. DOI: 10.14313/jamris/4-2019/40.

[7] Sadolewski J., Trybus B., Compiler and virtual machine of a multiplatform control environment. *Bulletin of the Polish Academy of Sciences Technical Sciences*, t. 70, nr 2, s. 1–9, 2022. DOI: 10.24425/bpasts.2022.140554.

[8] Beremiz: Integrated development environment. Dostęp online: http://www.beremiz.org. [09.09.2025].

[9] Tisserant E., Bessard L., De Sousa M., An open source IEC 61131-3 integrated development environment. [W:] *5th IEEE Int. Conf. on Industrial Informatics (INDIN)*, Vienna, s. 183–187, 2007. DOI: 10.1109/indin.2007.4384753

[10] GEB Automation: GEB Automation IDE Guide. Dostęp online: http://www.gebautomation.org. [09.09.2025].

[11] Keil: Embedded Development Tools. Dostęp online: https://www.keil.com/. [09.09.2025].

[12] Lindholm T., Yellin F., Bracha G., Buckley A., *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 2014. ISBN: 978-0133922721.

[13] Richter J., *CLR via C#*. Microsoft Press, 2012.

[14] Simros M., Wollschlaeger M., Theurich S., Programming embedded devices in IEC 61131-languages with industrial PLC tools using PLCopen XML. [W:] *CONTROLO 2012 - 10th Portuguese Conf. on Automatic Control*, Funchal, s. 51–55, 2012.

[15] Cavalieri S., Puglisi G., Scroppo M. S., Galvagno L., Moving IEC 61131-3 applications to a computing framework based on CLR virtual machine. [W:] *2016 IEEE 21st Int. Conf. on Emerging Technologies and Factory Automation (ETFA)*, Berlin, s. 1–8, 2016. DOI: 10.1109/etfa.2016.7733632.

[16] Kim H. S., Lee J. Y., Kwon W. H., A compiler design for IEC 1131-3 standard languages of programmable logic controllers. [W:] *SICE '99, 38th SICE Annual Conf.,* Morioka, s. 1155–1160, 1999. DOI: 10.1109/SICE.1999.788715.

[17] Milik A., Multiple core PLC CPU implementation and programming. *Journal of Circuits, Systems and Computers*, t. 27, nr 10, art. 1850162, s. 1–30, 2018. DOI: 10.1142/S0218126618501621.

[18] Milik A., Hrynkiewicz E., On Translation of LD, IL and SFC Given According to IEC-61131 for Hardware Synthesis of Reconfigurable Logic Controller. [W:] *IFAC Proceedings Volumes*, Cape Town, t. 19, nr 3, s. 4477–4483, 2014. DOI: 10.3182/20140824-6-ZA-1003.01333.

[19] Rzońca D., Sadolewski J., Stec A., Świder Z., Trybus B., Trybus L., Mini-DCS system programming in IEC 61131-3 structured text. *Journal of Automation, Mobile Robotics and Intelligent Systems*, t. 2, nr 3, s. 48–54, 2008.

[20] Zhang M., Lu Y., Xia T., The Design and Implementation of Virtual Machine System in Embedded SoftPLC System. [W:] *2013 Int. Conf. on Computer Sciences and Applications*, Wuhan, s. 775–778, 2013. DOI: 10.1109/CSA.2013.185.

[21] Patterson D. A., Hennessy J. L., *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Elsevier, 2013. ISBN: 978-0-12-407726-3.

[22] Mazidi M. A., Naimi Sep., Naimi Ser., *The AVR Microcontroller and Embedded Systems: Using Assembly and C*. MicroDigitalEd, 2017. ISBN: 978-0997925968.

[23] Trybus B., Development and implementation of IEC 61131-3 Virtual Machine. *Theoretical and Applied Informatics*, t. 23, nr 1, s. 21–35, 2011. DOI: 10.2478/v10179-011-0002-z.

[24] Lumel: Urządzenia i systemy automatyki. Dostęp online: https://www.lumel.com.pl. [09.09.2025].

[25] Praxis Automation: Ship automation and navigation systems. Dostęp online: https://www.praxis-automation.nl/. [09.09.2025].

[26] iGrid T&D: Smart Solutions for Smart Grids. Dostęp online: https://www.igrid-td.com/. [09.09.2025].

[27] Bai Y.: *Practical Microcontroller Engineering with ARM Technologies*. Wiley-IEEE Press, 2015. ISBN: 978-1-119-05237-1.

[28] Upton E., Halfacree G., *Raspberry Pi User Guide*. Wiley Publishing, 2014. ISBN: 978-1-118-79548-4.

[29] Phoenix Contact: Automation Technology for PLCnext Technology. Dostęp online: https://www.phoenixcontact.com/en-gb/products/plcs-controllers-and-i-os/automation-technology-for-plcnext-technology. [09.09.2025].

[30] Beckhoff Automation: CX8200 Embedded PC Series compact PLC. Dostęp online: https://www.beckhoff.com/en-en/products/ipc/embedded-pcs/cx8200-arm-r-cortex-r-a53/ [09.09.2025].

[31] Pytel M., *Wysokowydajne sterowniki PLC Astraada One*. Poradnik Automatyka, nr 3, s. 25–30, 2023. Dostęp online: https://www.astor.com.pl/poradnikautomatyka/wysokowydajne-sterowniki-plc-astraada-one-modular-oparte-na-raspberry-pi/. [09.09.2025].

[32] Engel M., Lasota S., Tarlecki A., *Semantyka i weryfikacja programów*. Dostęp online: https://wazniak.mimuw.edu.pl/index.php?title=Semantyka_i_weryfikacja_program%C3%B3w#modu%C5%82y. [09.09.2025].

[33] Gordon M. J. C., *Denotacyjny opis języków programowania*. WNT, 1983.

[34] Hubacz M., Sadolewski J., Trybus B., Wydajność architektury STM32 w zakresie wykonywania kodu pośredniego dla systemów sterowania. *Pomiary Automatyka Robotyka*, t. 25, nr 1, s. 27–34, 2021. DOI: 10.14313/PAR_239/27.

[35] Hubacz M., Trybus B., Dual-Core PLC for Cooperating Projects with Software Implementation. *Electronics*, t. 12, nr 23, art. 4730, s. 1–16, 2023. DOI: 10.3390/electronics12234730.

[36] Norris D., *Programming with STM32: Getting Started with the Nucleo Board and C/C++*. McGraw-Hill Education TAB, 2018. ISBN: 978-1260031317.

[37] Trybus B., Trybus L., Tryb konfiguracji pewnego rozproszonego systemu sterowania. [W:] Madeyski L., Kosiuczenko P., Bolanowski M. (eds) *Inżyniera oprogramowania i systemy czasu rzeczywistego,* PTI, Rzeszów, s. 93-107, 2017. ISBN: 978-83-946253-3-7.

[38] STMicroelectronics: STM32H745/755 microcontrollers. Dostęp online: https://www.st.com/en/microcontrollers-microprocessors/stm32h745-755.html. [09.09.2025].

[39] Finkenzeller K., *RFID handbook*. Wiley, 2010. ISBN: 0-470-84402-7. DOI: 10.1002/9780470665121.

[40] WAVE ROVER: Waveshare Wiki. Dostęp online: https://www.waveshare.com/wiki/WAVE_ROVER. [09.09.2025].

[41] Dogan I., Ahmet I., *The Official ESP32 Book*. Elektor International Media, 2017. ISBN: 978-1907920639.

[42] Hubacz M., Pawłowicz B., Trybus B., Using Multiple RFID Readers in Mobile Robots for Surface Exploration. [W:] *Automation 2019: Adv. in Intell. Syst. and Computing,* Springer, t. 920, s. 469–480, 2020. DOI: 10.1007/978-3-030-13273-6_42.

[43] Jamro M., Trybus B., IEC 61131 3 programmable human machine interfaces for control devices. [W:] *6th Int. Conf. on Human System Interactions (HSI)*, Sopot, s. 48–55, 2013. DOI: 10.1109/HSI.2013.6577801.

[44] Rockwell Automation: ISaGRAF Workbench. Dostęp online: http://www.isagraf.com. [09.09.2025].

[45] STRATON: Industrial software for automation control. Dostęp online: http://www.straton-plc.com. [09.09.2025].

[46] Shin S., Kwon M., Rho S., Whimori CDK: A Control Program Development Kit. [W:] *Eng. and Info. 2009 Int. Conf. on Computing*, Fullerton, s. 115–118, 2009. DOI: 10.1109/ICC.2009.3.

[47] De Tommasi G., Pironti A., An educational open-source tool for the design of IEC 61131-3 compliant automation software. [W:] *2008 Int. Symp. on Power Electronics, Electrical Drives, Automation and Motion*, Ischia, s. 486–491, 2008. DOI: 10.1109/SPEEDHAM.2008.4581144.

[48] Hubacz M., Trybus B., Data Alignment on Embedded CPUs for Programmable Control Devices. *Electronics*, t. 11, nr 14, art. 2174, s. 1–16, 2022. DOI: 10.3390/electronics11142174.

[49] Cooper K. D., Torczon L., *Engineering a compiler*. Morgan Kaufmann Publishers, 2022. ISBN: 978-0128154120.

[50] Ferreira E., Paulo R., Da Cruz D., Henriques P., Integration of the ST language in a model-based engineering environment for control systems. *Computer Science and Information Systems*, t. 5, nr 2, s. 87–101, 2008. DOI: 10.2298/csis0802087f.

[51] Rzońca D., Trybus B., Hierarchical Petri Net for the CPDev Virtual Machine with Communications. [W:] Kwiecień, A., Gaj, P., Stera, P. (eds) *Comp. Networks: Commun. in Comp. and Info. Sci.*, t. 39, Springer, 2009. DOI: 10.1007/978-3-642-02671-3_31.

[52] Sadolewski J., Stec A., Programowany sterownik CPCtrl z kartą I/O. *Napędy i Sterowanie*, t. 11, nr 139, s. 72–74, 2010. ISBN: 1507-7764.

[53] CPDev-Control Program Developer. Dostęp online: https://github.com/CPDev-ControlProgramDeveloper. [09.09.2025].

[54] Sadolewski J., Trybus B., Denotational model and implementation of scalable Virtual Machine in CPDEV. [W:] *2022 17th Conf. on Comp. Sci. and Intelligence Systems (FedCSIS)*, Sofia, s. 587–591, 2022. DOI: 10.15439/2022F236.

[55] Sadolewski J., Trybus B., Exception Handling in Programmable Controllers with Denotational Model. [W:] *2023 18th Conf. on Comp. Sci. and Intelligence Systems (FedCSIS)*, Warszawa, s. 721–730, 2023. DOI: 10.15439/2023F5651.

[56] Syme D., Granicz A., Cisternino A., *F# 4.0 dla zaawansowanych. Wydanie IV*, Helion, 2017. ISBN: 978-83-283-2943-0.

[57] Siemens: S7-PLCSIM. Dostęp online: https://mall.industry.siemens.com/mall/pl/pl/Catalog/Products/10316003. [09.09.2025].

[58] Cisek J., Mikluszka W., Świder Z., Trybus L., A Low-Cost DCS with Multifunction Instruments and CAN Bus. *IFAC Proceedings Volumes*, t. 34, nr 29, 2001, s. 64–69. DOI: 10.1016/S1474-6670(17)32794-5.

[59] *Global ARM Microcontrollers Market Size By Product, By Application, By Geographic Scope And Forecast*, 2024.
Dostęp online: https://www.verifiedmarketresearch.com/product/arm-microcontrollers-market/. [09.09.2025].

[60] *Xtensa LX Microprocessor Overview Handbook*. Cadence Design Systems, 2004.

[61] Patterson D., Waterman A., *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017. ISBN: 978-0999249116.

[62] Sun J., Jones M., Reinauer S., Zimmer V., *Embedded Firmware Solutions*. Apress, 2015. ISBN: 978-1-4842-0070-4.

[63] Alvarez M., Salami E., Ramirez A., Valero M., Performance impact of unaligned memory operations in SIMD extensions for video codec applications. [W:] *Proc. of the 2007 IEEE Int. Symp. on Performance Analysis of Systems & Software*, San Jose, s. 62–71, 2007. DOI: 10.1109/ISPASS.2007.363737.

[64] Pinheiro R. L., Landa-Silva D., Qu R., Constantino A. A., Yanaga E., An application programming interface with increased performance for optimisation problems data. *Journal of Management Analytics*, t. 3, nr 4, s. 305–332, 2016. DOI: 10.1080/23270012.2016.1233514.

[65] Stec A., Ship Maneuvering Model for Autopilot Simulator. [W:] Szewczyk, R.; Zieliński, C.; Kaliczyńska, M. (red.): *Progress in Automation, Robotics and Measuring Techniques. ICA 2015*, Springer, t. 350, s. 305–332, 2015. DOI: 10.1007/978-3-319-15796-2_27.

[66] *OS X ABI Mach-O File Format Reference*. Apple, 2009. Dostęp online: https://www.symbolcrash.com/wp-content/uploads/2019/02/ABI_MachOFormat.pdf . [09.09.2025].

[67] Wu H., Chen C., Weng K., An Energy-Efficient Strategy for Microcontrollers. *Applied Sciences,* t. 11, nr 6, art. 2581, s. 1–18, 2021. DOI: 10.3390/app11062581.

[68] Wang K. C., *Embedded Real-Time Operating Systems.* Springer, s. 401–475, 2017. DOI: 10.1007/978-3-031-28701-5.

[69] ABB: Freelance – Stacje procesowe. Dostęp online: https://new.abb.com/control-systems/pl/essential-automation/freelance/stacje-procesowe. [09.09.2025]

[70] Beckhoff: IPC – Tabular product overview. Dostęp online: https://www.beckhoff.com/en-en/products/ipc/tabular-product-overview/. [09.09.2025].

[71] CODESYS Group. Dostęp online: https://www.codesys.com/. [09.09.2025].

[72] The OpenAMP Project. Dostęp online: https://www.openampproject.org/. [09.09.2025].

[73] PLCopen: *PLCopen Document, Technical Paper, PLCopen Promotional Committee Training: Software Construction Guidelines initiative*, 2016.

Dostęp online: https://www.plcopen.org/download_file/force/ff60e817-eee5-442d-b718-a357a7279c7e/342//. [09.09.2025].

[74] Hajduk Z., Trybus B., Sadolewski J., Architecture of FPGA Embedded Multiprocessor Programmable Controller. *IEEE Transactions on Industrial Electronics*, t. 62, nr 5, s. 2952–2961, 2015. DOI: 10.1109/TIE.2014.2362888.

[75] Milik A., Walichiewicz M., Shared-Semaphored Cache Implementation for Parallel Program Execution in Multi-Core Systems. *International Journal of Electronics and Telecommunications,* t. 69, nr 2, s. 371–382, 2023. DOI: 10.24425/ijet.2023.144373.

[76] Jiang W., Sun L., Chen Y., Ma H., Hashimoto S., A Hardware-in-the-Loop-on-Chip Development System for Teaching and Development of Dynamic Systems. *Electronics*, t. 10, nr 7, art. 801, s. 1–15, 2021. DOI: 10.3390/electronics10070801.

[77] Pytel M., *Sterowniki PLC Astraada One: czy opłaca się inwestować w jednostki centralne z procesorem dwurdzeniowym?* Poradnik Automatyka, nr 2, s. 28–30, 2023. Dostęp online: https://www.astor.com.pl/poradnikautomatyka/sterowniki-plc-astraada-one-czy-oplaca-sie-inwestowac-w-jednostki-centralne-z-procesorem-dwurdzeniowym/. [09.09.2025].

[78] Beckhoff: Multi-task data access synchronization in the PLC. Dostęp online: https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/45844579955484184843.html&id=. [09.09.2025].

[79] Stój J.: *Wybrane zagadnienia sieci komunikacyjnych w przemysłowych systemach komputerowych*. Wydawnictwo Politechniki Śląskiej, 2023. ISBN: 9788378809159.

[80] Olivas Mendoza J. R., *Developing Windows Services Succinctly*. Syncfusion Inc., 2015. ISBN: 978-1-64200-089-4.

[81] Hubacz M., Pawłowicz B., Trybus B., Exploring a Surface Using RFID Grid and Group of Mobile Robots. [W:] *Automation 2018 Adv. in Intell. Syst. and Computing*, Springer, t. 743, s. 490–499, 2018. DOI: 10.1007/978-3-319-77179-3_46.

[82] Pawłowicz B., Skoczylas M., Trybus B., Salach M., Hubacz M., Mazur D., Navigation and mapping of closed spaces with a mobile robot and RFID grid. *Archives of Control Sciences*, t. 33, nr 4, s. 737–759, 2023. DOI: 10.24425/acs.2023.148879.

[83] Thepsit T., Konghuayrob P., Saenthon A., Yanyong S., Localization for Outdoor Mobile Robot Using LiDAR and RTK-GNSS/INS. *Sensors and Materials*, t. 36, nr 4, art. 1405, 2024. DOI: 10.18494/SAM4841.

[84] Beckhoff: TF7XXX TC3 Vision. Dostęp online: https://infosys.beckhoff.com/english.php?content=../content/1033/tf7xxx_tc3_vision/6917339659.html&id=. [09.09.2025].

[85] Abbott D., *Linux for Embedded and Real-Time Applications*. Mewmes, 2012. ISBN: 978-0124159969.

[86] Reghenzani F., Massari G., Fornaciari W. The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *ACM Computing Surveys (CSUR),* t. 52, nr 1, art. 18, s. 1–36, 2020. DOI: 10.1145/3297714.

[87] Longo E., Redondi A. E.C., Design and implementation of an advanced MQTT broker for distributed pub/sub scenarios. *Computer Networks*, t. 224, art. 109601, 2023. DOI: 10.1016/j.comnet.2023.109601.

[88] Munarto R., Wiryadinata R., Utama D. P., *Implementation of AHRS (Attitude Heading and Reference Systems) With Madgwick Filter as Hexapods Robot Navigation*. [W:] *2022 Int. Conf. on Informatics Electrical and Electronics (ICIEE)*, Yogyakarta, s. 1–9, 2022. DOI: 10.1109/ICIEE55596.2022.10010044.