POZNAN UNIVERSITY OF TECHNOLOGY ACADEMIC JOURNALS
No 99 Electrical Engineering 2019

DOI 10.21008/j.1897-0737.2019.99.0014

Agnieszka CHOROSZUCHO*, Piotr GOLONKO*, Mateusz SUMOREK*
Jakub ŻUKOWSKI*

PORÓWNANIE WYDAJNOŚCI WYSOKOPOZIOMOWYCH JĘZYKÓW PROGRAMOWANIA W SYSTEMACH MIKROPROCESOROWYCH

Prowadzenie prac naukowych, zwłaszcza z dziedzin technicznych często związane jest z koniecznością wykonania prototypu urządzenia lub potrzebnych przyrządów. W takich przypadkach często wykorzystuje się rozwiązania oparte na mikrokontrolerach. Wybór samej platformy sprzętowej, jak również programistycznej decyduje o sukcesie projektu oraz pozwala zaoszczędzić czas. W artykule przedstawiono wyniki badań dotyczące trzech platform opartych o różne modele mikrokontrolerów oraz zastosowane dwa języki programowania (C/C++ oraz nowe rozwiązanie bazujące na języku Python). Do analizy wydajności języków programowania wybrano zagadnienia bazujące na procedurach matematycznych mających zastosowanie w sterowaniu, analizie i automatyce. Wyniki badań pozwolą wpłynąć na właściwy dobór optymalnej platformy sprzętowej oraz języka programowania przy uwzględnieniu planowanego zastosowania i zapotrzebowania na moc obliczeniową.

SŁOWA KLUCZOWE: mikrokontrolery, C/C++, Phyton, Arduino, ESP32, ESP8266.

1. WPROWADZENIE

Prace naukowe często związane są z potrzebą implementacji opracowanych rozwiązań na platformie sprzętowej w postaci modułów z mikrokontrolerami. W ostatnich latach liczba platform ograniczała się do 2-3 popularnych rozwiązań z pełną dokumentacją techniczną oraz dużym wsparciem społeczności internetowej. Platformy typu: AVR oraz PIC czy również rodzina 8051 pomimo ciągłego rozwijania przez producentów nie mogą zapewnić odpowiedniej mocy obliczeniowej a sama architektura 8-bitowa stanowi coraz większe ograniczenie w ilości obsługiwanej pamięci oraz szybkich peryferii.

Ostatnio rynek mikrokontrolerów wzbogacił się o nowe rozwiązania, które zostały entuzjastycznie przyjęte przez konstruktorów, ponieważ posiadają kompletną dokumentację oraz wsparcie producentów, jak również społeczności in-

* Politechnika Białostoka

ternetowej. Są to mikrokontrolery bazujące na architekturze 32-bitowej opartej o rdzeń ARM [9]. Główne zalety, to wydajna jednostka obliczeniowa wyposażone w niedostępną dla 8-bitowców ilość pamięci oraz szybki interfejs w tym łączność bezprzewodowa LoRa, WiFi oraz również BLE.

Przy uwzględnieniu powyższych, pojawia się problem związany z wyborem platformy, gdzie często starsze platformy jak Arudino [8] bazujące na 8-bitowym mikrokontrolerze AVR mają ugruntowaną pozycję na rynku, ogromną ilość dokumentacji, przykładów, bibliotek oraz duże wsparcie środowiska konstruktorów. Również środowisko programistyczne jest stabilne i dojrzałe do profesjonalnych rozwiązań. Z drugiej strony platformy 32-bitowe są rozwojowe, wydajne a dokumentacja techniczna jest zazwyczaj więcej niż zadawalająca. Należy podkreślić, że środowiska programistyczne są coraz bardziej stabilne i rozbudowane.

Ze względu na wady i ograniczenia architektury 8-bitowej można przyjąć, iż starsze rozwiązania opierające się o architekturę 8-bitową są systematycznie wypierane przez 32-bitowe rozwiązania bazujące głównie na architekturze ARM [3, 7]. Naturalną konsekwencją popularyzacji platform 32-bitowych jest coraz mniejszy udział programów pisanych w języku Asembler na rzecz języków wyższego poziomu takich jak C czy C++ [2] oraz microPython [1, 4]. Języki C oraz C++ są powszechnie stosowane na części mikrokontrolerów 8-bitowych, natomiast język microPython jest nadal na nich nie wykorzystywany.

W celu ułatwienia wdrażania rozwiązań opartych o mikrokontrolery w artykule zostało przedstawiona analiza wydajności platform mikrokontrolerowych oraz dwa języki programowania: C/C++ i microPython. Wybór języka C/C++ jest oczywisty, natomiast zastosowanie języka Python [5] może być poddane w wątpliwość, ponieważ w przeciwieństwie do C/C++ jest językiem interpretowanym, a nie kompilowanym, co może w znaczący sposób wpłynąć na wydajność programu. Należy podkreślić, że język Python ma szereg zalet m.in.: szybkość nauki języka, dużą ilość bibliotek czy brak potrzeby kompilowania. Możliwe jest również przenoszenie znacznej części kodu z komputera klasy PC na mikrokontroler, co między innymi pozwala na przetestowanie algorytmu na PC, a następnie szybkie jego przeniesienie na dowolny mikrokontroler.

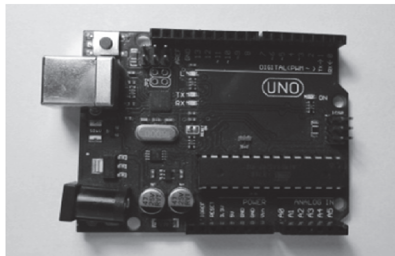
2. PLATFORMY TESTOWE

W dobie popularyzacji techniki mikroprocesorowej oraz dynamicznego jej rozwoju duże zainteresowanie znalazły tanie platformy do tzw. błyskawicznego prototypowanie (*rapid prototyping*). Do analizy zastosowano płytkę Arduino Uno wyposażoną w układ Atmega328p firmy Atmel (rys. 1), płytkę NodeMCU posiadającą moduł ESP8266MOD firmy Espressif Systems (rys. 2) oraz płytkę rozwojową LOLIN D32 wyposażoną w moduł ESP32-WROVER również firmy Espressif Systems (rys. 3). Układ Atmega328p jest 8-bitowym procesorem pra-

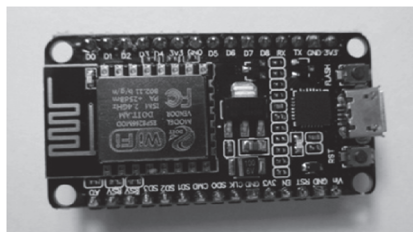
cującym z taktowaniem zegara o częstotliwości 16 MHz zaś układ ESP8266 jest oparty na 32-bitowym procesorze Xtensa Diamond Standard 106 Micro firmy Tensilica pracujący z częstotliwością 80 MHz [7]. Trzeci układ oparty jest o 32-bitowy procesor Xtensa LX6 również firmy Tensilica [3], który pracuje z częstotliwością zegara 240 MHz.

Wydajność obu rozpatrywanych języków programowania została przeprowadzona na testowanych platformach sprzętowych przy uwzględnieniu czasu wykonania zadania. Do rozpatrywanych zadań testowych zaliczono:

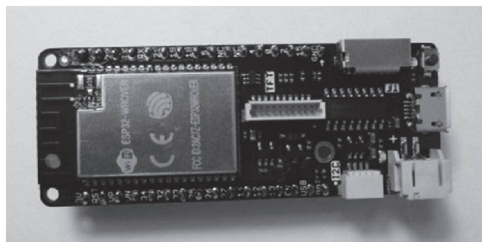
– obliczenie liczby π z dokładnością do 6 miejsc po przecinku,
– obliczenie 10000 wyznaczników losowo wygenerowanych macierzy o wymiarach 4×4.



Rys. 1. Płytka Arduino Uno



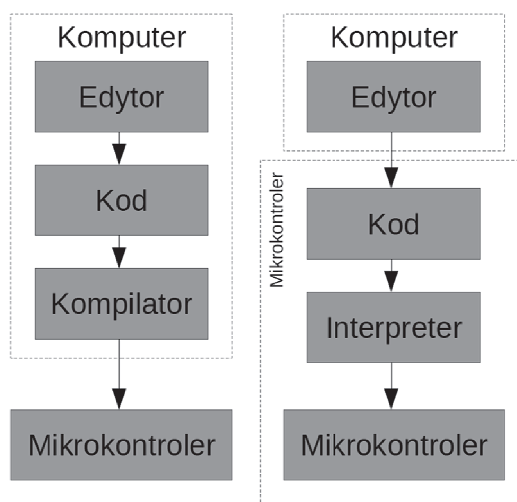
Rys. 2. Płytka NodeMCU



Rys. 3. Płytka Lolin D32

Tabela 1. Zestawienie platform testowych.

Płytka	Procesor	RAM	Flash	Taktowanie	Architektura
Arduino UNO	Atmega328P	2 KB	32 KB	16 MHz	8-bit
NodeMCU	ESP8266	96 KB	4 MB	80 MHz	32-bit
Lolin D32	ESP-32	320 KB	4 MB	240 MHz	32-bit



Rys. 4. Schematy przedstawiające sposób implementacji kodu (lewa strona rysunku – język C, prawa strona – język Python

3. PRZYJĘTE ZAŁOŻENIA

W celu obliczenia liczby π został zastosowany wzór (1) Gottfrieda Wilhelma Leibniza. Należy podkreślić, że nie jest to wydajny sposób obliczania liczby π przy użyciu narzędzi numerycznych, ponieważ ta metoda jest wolno zbieżna [6]. Wspomniana metoda wymaga dużej ilości obliczeń w celu osiągnięcia akceptowalnego przybliżenia liczby π. Z tego powodu jest adekwatna w przeprowadzonym badaniu, gdzie celem jest sprawdzenie wydajności obliczeniowej (ilość operacji w czasie) urządzenia podczas wymuszenia dużej ilości operacji arytmetycznych.

$$\pi = 4 \cdot \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{2n-1} = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots\right) \qquad (1)$$

Drugim zagadnieniem wykorzystanym w analizie wydajności było obliczenie 10000 wyznaczników za pomocą wzoru (2) pseudolosowo wygenerowanych macierzy kwadratowych o wymiarach 4×4. Każdy element macierzy przyjmował wartość z zakresu od 0 do 4096. W tym celu zastosowano wzór (2):

$$\det\left|\mathbf{A}\right| = \det\begin{bmatrix} a_{11} & \cdots & a_{14} \\ \vdots & \ddots & \vdots \\ a_{41} & \cdots & a_{44} \end{bmatrix} \tag{2}$$

W celu napisania algorytmu obliczania wyznaczników macierzy został użyty wzór Leibniza (3) dla wyznaczników. Wzór został wykorzystany do testów. Na co dzień nie stosuje się tego wzoru (3), ponieważ jest nieefektywny dls macierzy większych od 3×3.

$$\det(A) = \sum_{\tau \in S_n} \operatorname{sgn}(\tau) \prod_{i=1}^{n} a_{i,\tau(i)} = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i(i),i} \tag{3}$$

gdzie: **A** – macierz o wymiarach $n \times n$, det(A) - wyznacznik macierzy **A**, a_{ij} – element macierzy w wierszu (i) oraz kolumnie (j), sgn(σ) – funkcja znaku.

4. ALGORYTMY OBLICZENIOWE

```
import urandom
import time
row = 0
column = 0
value = 0
m = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
start = time.ticks_ms()
for matrix in range(10000):
  for i in range(16):
    value = urandom.getrandbits(12)
    if row==4:
        row=0
        column += 1
    if column==4: column = 0
    m[column][row] = value
    row += 1
    determinant = (
      m[0][3] * m[1][2] * m[2][1] * m[3][0] - m[0][2] * m[1][3] * m[2][1] * m[3][0] -
      m[0][3] * m[1][1] * m[2][2] * m[3][0] + m[0][1] * m[1][3] * m[2][2] * m[3][0] +
      m[0][2] * m[1][1] * m[2][3] * m[3][0] - m[0][1] * m[1][2] * m[2][3] * m[3][0] -
      m[0][3] * m[1][2] * m[2][0] * m[3][1] + m[0][2] * m[1][3] * m[2][0] * m[3][1] +
      m[0][3] * m[1][0] * m[2][2] * m[3][1] - m[0][0] * m[1][3] * m[2][2] * m[3][1] -
      m[0][2] * m[1][0] * m[2][3] * m[3][1] + m[0][0] * m[1][2] * m[2][3] * m[3][1] +
      m[0][3] * m[1][1] * m[2][0] * m[3][2] - m[0][1] * m[1][3] * m[2][0] * m[3][2] -
      m[0][3] * m[1][0] * m[2][1] * m[3][2] + m[0][0] * m[1][3] * m[2][1] * m[3][2] +
      m[0][1] * m[1][0] * m[2][3] * m[3][2] - m[0][0] * m[1][1] * m[2][3] * m[3][2] -
      m[0][2] * m[1][1] * m[2][0] * m[3][3] + m[0][1] * m[1][2] * m[2][0] * m[3][3] +
      m[0][2] * m[1][0] * m[2][1] * m[3][3] - m[0][0] * m[1][2] * m[2][1] * m[3][3] -
      m[0][1] * m[1][0] * m[2][2] * m[3][3] + m[0][0] * m[1][1] * m[2][2] * m[3][3])
end = time.ticks_ms()
print(end - start)
print(determinant)
```

Rys. 5. Algorytm generowania macierzy oraz obliczania jej wyznaczników przy wykorzystaniu języka MicroPython

```
while (count < 1):
  start = time.ticks_ms()
  a=0

  for i in range (250000):
    a += ((-1)**i)/(2*i+1)

  end = ticks_ms()
  print(end-start)
  print(4*a)
  count = 1
```

Rys. 6. Algorytm obliczania wartości liczby **π** przy wykorzystanu języka MicroPython

```
int m[4][4] = {
               {0,0,0,0},
               {0,0,0,0},
               {0,0,0,0},
               {0,0,0,0}
               };
Serial.println(millis());
for(unsigned long start = 0; start<10000;start++)
{

  for(uint8_t i =0; i<16; ++i)
    {
      value = random(0,4096);
      if(row == 4)
      {
        row=0;
        column++;
      }
      if(column==4) column=0;
      m[column][row] = value;
      row++;
      cnt++;
      determinant = (
       m[0][3] * m[1][2] * m[2][1] * m[3][0] - m[0][2] * m[1][3] * m[2][1] * m[3][0] -
       m[0][3] * m[1][1] * m[2][2] * m[3][0] + m[0][1] * m[1][3] * m[2][2] * m[3][0] +
       m[0][2] * m[1][1] * m[2][3] * m[3][0] - m[0][1] * m[1][2] * m[2][3] * m[3][0] -
       m[0][3] * m[1][2] * m[2][0] * m[3][1] + m[0][2] * m[1][3] * m[2][0] * m[3][1] +
       m[0][3] * m[1][0] * m[2][2] * m[3][1] - m[0][0] * m[1][3] * m[2][2] * m[3][1] -
       m[0][2] * m[1][0] * m[2][3] * m[3][1] + m[0][0] * m[1][2] * m[2][3] * m[3][1] +
       m[0][3] * m[1][1] * m[2][0] * m[3][2] - m[0][1] * m[1][3] * m[2][0] * m[3][2] -
       m[0][3] * m[1][0] * m[2][1] * m[3][2] + m[0][0] * m[1][3] * m[2][1] * m[3][2] +
       m[0][1] * m[1][0] * m[2][3] * m[3][2] - m[0][0] * m[1][1] * m[2][3] * m[3][2] -
       m[0][2] * m[1][1] * m[2][0] * m[3][3] + m[0][1] * m[1][2] * m[2][0] * m[3][3] +
       m[0][2] * m[1][0] * m[2][1] * m[3][3] - m[0][0] * m[1][2] * m[2][1] * m[3][3] -
       m[0][1] * m[1][0] * m[2][2] * m[3][3] + m[0][0] * m[1][1] * m[2][2] * m[3][3]);
    }
}
  Serial.println(millis());
  Serial.println(determinant);
```

Rys. 7. Algorytm napisany w języku C, służący do generowania macierzy oraz obliczania jej wyznaczników

```
Serial.println(millis());
float pi_value = 4;
int flag = 0;
unsigned long cnt;
for(unsigned long i=3; i<500000; i+=2){
  cnt++;
  if(flag){
    pi_value = pi_value + (4.0/i);
  }
  else{
    pi_value = pi_value - (4.0/i);
  }
 flag= !flag;
}
Serial.println(millis());
Serial.println(pi_value, 6);
```

Rys. 8. Algorytm obliczania wartości liczby **π** przy wykorzystaniu języka C

5. WYNIKI TESTÓW

Na podstawie uzyskanych pomiarów można jednoznacznie stwierdzić, że MicroPython nie sprawdza się w zadaniach wymagających intensywnych obliczeń. Można zauważyć, że język C jest nieporównywalnie szybszy od MicroPython, zwłaszcza przy zadaniu takim jak obliczanie wyznaczników macierzy (tabela 3 oraz tabela 4).

Tabela 2. Wyniki pomiarów dla Arduino Uno.

	Arduino UNO	
Język	C	MicroPython
Taktowanie zegara [MHz]	16	
Ilość iteracji	250 000	Nie dotyczy
Czas obliczania π [ms]	10 776	Nie dotyczy
Obliczona wartość π	3.141592	Nie dotyczy
Ilość obliczonych macierzy	10 000	Nie dotyczy
Czas obliczania wyznaczników macierzy [ms]	14 760	Nie dotyczy

Tabela 3. Wyniki pomiarów dla NodeMCU.

	NodeMCU	
Język	C	MicroPython
Taktowanie zegara [MHz]	80	
Ilość iteracji	250 000	250 000
Czas obliczania π [ms]	3 144	43 505
Obliczona wartość π	3.141591	3,051740
Ilość obliczonych macierzy	10 000	10 000
Czas obliczania wyznaczników macierzy [ms]	794	1 287 271 (~21 min)

W celu znalezienia przyczyny długiego czasu wykonywania obliczeń w języku MicroPython, zostało dodatkowo wykonane badanie mające na celu sprawdzenie czy generowanie losowych macierzy wpływa znacząco na wykonywane algorytmu. Wyniki testu ujawniają, że w przypadku ESP8266 czas generacji wynosi 54148 ms, zaś przy ESP32 11174 ms (tabela 5). Zatem można przyjąć, że czas jest na tyle krótki, że nieznacznie wpływa na czas wykonania algorytmu.

Tabela 4. Wyniki pomiarów dla LOLIN D32.

	LOLIN D32	
Język	C	MicroPython
Taktowanie zegara [MHz]	240	
Ilość iteracji	250 000	250 000
Czas obliczania π [ms]	827	5895
Obliczona wartość π	3,141592	3,141592
Ilość obliczonych macierzy	10 000	10 000
Czas obliczania wyznaczników macierzy [ms]	182	129 682

Tabela 5. Wyniki pomiarów czasu generowania macierzy.

Platforma testowa	NodeMCU	LOLIN D32
Czas generowania macierzy MicroPython [ms]	54148	11174

6. PODSUMOWANIE

W artykule przedstawiono wyniki pomiarów wydajności wysokopoziomowych języków programowania z wykorzystaniem różnych platform testowych. Celem zastosowanych algorytmów było sprawdzenie, jak testowane języki programowania radzą sobie z zadaniami wymagającymi dużej ilości obliczeń. Badania będą kontynuowane z użyciem innych powszechnie testowanych platform rozwojowych oraz bibliotek specjalizujących się w określonych zadaniach.

LITERATURA

[1] Tollervey N.H., Programming with MicroPython: Embedded Programming with Microcontrollers and Python, O'Reilly Media, 2017.
[2] Oualline S., Practical C programming, O'Reilly Media, 1997.
[3] Espressif Systems. (grudzień 2018). espressif.com [Online]. Dostępne: https://espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf, (Na dzień: 20 stycznia 2019).
[4] Norris D., Python for Microcontrollers: Getting Started with MicroPython, McGraw-Hill Education TAB, 2017.

[5] Martelli A., Ascher D., Martelli Ravenscroft A., Python Cookbook, Third edition, O'Reilly Media, 2013.
[6] Borwein J.M., Borwein P.B., Bailey D.H., The American Mathematical Monthly, Vol. 96, No. 3, pp. 201–219, Mathematical Association of America, (Mar., 1989).
[7] Espressif Systems. (grudzień 2018). espressif.com [Online]. Dostępne: https://www.espressif.com/sites/default/files/documentation/0a-esp8285_datasheet_en.pdf, (Na dzień: 20 stycznia 2019).
[8] Syed Omar Faruk Towaha, Learning C for Arduino, Packt Publishing, 2017.
[9] Yiu J., The Definitive Guide to the ARM Cortex-M0, Newnes, 2011.

Acknowledgment. This work was prepared under scientific work S/WE/2/18 and supported by the Polish Ministry of Science and Higher Education.

COMPARISON OF HIGH-LEVEL PROGRAMMING LANGUAGES EFFICIENCY IN EMBEDDED SYSTEMS

Conducting research, especially in technical field often binds with necessity of making device prototype or specialized tools. In such situations microcontroller-based solutions are often used. Choice of development platform and software environment decides about success of project or allows saving significant amount of time. In article have been tested 3 common development platforms based on different microcontrollers and two high-level programming languages, C/C++ and new solution based on Python. Chosen testing process is based on mathematical procedures used in control, analysis and automation. Results of research should allow to selection of optimal hardware platform as well programming language according to planned use and requested computing power.

(Received: 04.02.2019, revised: 06.03.2019)