

# PPMGR Raport

Jakub Ostrzołek

## Problem

Na ten moment do użytku dostępne jest wiele języków programowania, które w swoim zamyśle mają mieć możliwość zastosowania w układach wbudowanych; np. Rust, Go czy Zig. Mimo tego, wciąż znacząca większość kodu pisanego pod systemy wbudowane jest w C lub C++.

W pracy postaram się porównać pod różnymi kątami trzy języki: C, Rust i Zig, by przeanalizować czy, a jeśli tak to w jakich przypadkach warto skorzystać z nowszych z nich.

C jest oczywiście na ten moment standardem w sektorze systemów wbudowanych i prawdopodobnie jeszcze całkiem długo tak właśnie pozostanie. Rust i Zig to dużo młodsze języki, które wykorzystują nowoczesne paradygmaty programowania i bardziej wspomagają programistę w tworzeniu poprawnego kodu. Pomimo tego, że są to języki wyższego poziomu niż C, to również umożliwiają wydajne operacje niskopoziomowe, dzięki czemu powinny znaleźć zastosowanie w układach wbudowanych.

Jako platformy sprzętowej planuję użyć w pierwszej kolejności jednego z układów z rodziny Raspberry PI, ponieważ można na nim łatwo uruchomić system typu UNIX, do którego jestem najbardziej przyzwyczajony. Jeżeli wystarczy czasu, chciałbym przeprowadzić testy również na jednym z układów z rodziny ESP-32.

## Sposób porównania języków

W celu ilościowej oceny poszczególnych języków, posłużę się metrykami przedstawionymi poniżej.

- Czas wykonania – w celu zmierzenia wydajności, która może mieć kluczowe znaczenie chociażby w systemach czasu rzeczywistego. Można mierzyć bezpośrednio w kodzie aplikacji lub skorzystać z narzędzia jak np. `valgrind --tool=callgrind`. Narzędzie działa na pewno dla programów napisanych w C i w Rust (`iai_callgrind`), do sprawdzenia czy działa w Zig.
- Zużycie pamięci operacyjnej. Manualne mierzenie w kodzie będzie bardzo skomplikowane. W C i Rust dostępne jest narzędzie `valgrind --tool=massif`, w Zig do sprawdzenia.

- Liczba linii kodu (jednostka kLOC – kilo line of code). Metryka pozwoli ukazać w przybliżeniu jak wysokopoziomowy jest kod. Języki niskopoziomowe wymagają z reguły pisania wielu, prostych linii kodu (np. assembler), a języki wysokopoziomowe niewielu linii kodu wykonujących bardziej skomplikowane operacje (np. Python). Te drugie są zazwyczaj preferowane przez programistów ze względu na większą czytelność i krótszy czas pisania, jednak te pierwsze co do zasady umożliwiają bardziej wydajne implementacje. Programy o wysokiej wydajności i niskim kLOC będą więc preferowane.
- Średnia złożoność cyklometryczna funkcji. Bardziej niskopoziomowe języki często mają większą złożoność cyklometryczną, co redukuje czytelność kodu. Mniejsza złożoność cyklometryczna funkcji będzie więc preferowana.
- Miary złożoności Halstead – analogicznie jak punkt powyżej.
- Wielkość pliku wykonywalnego – raczej nie jest najbardziej istotnym kryterium w obecnych czasach, jednak w niszowych sytuacjach może mieć znaczenie (np. dla rozwijającego się obszaru mikrorobotyki w medycynie, gdzie każdy mikrometr fizycznego rozmiaru może mieć znaczenie).

Oprócz tego będę porównywał mniej mierzalne aspekty takie jak:

- dostępność bibliotek ułatwiających realizację zadań,
- łatwość korzystania z systemu do budowania aplikacji (wbudowanego w ekosystem języka lub zewnętrznego), w tym kompilowania skrośnego na docelową platformę,
- czytelność kodu (porównanie fragmentów kodu odpowiedzialnego za podobną funkcjonalność w różnych językach),
- błędy działania programu powstałe podczas jego tworzenia, niewykryte przez kompilator albo analizator kodu (np. przepełnienie bufora, pisanie do niezaalokowanej pamięci, itp; bez błędów typowo logicznych),
- kod pośredni LLVM dla prostszych przykładów, celem znalezienia interesujących różnic między kompilatorami danych języków.

## Zadania

1. Program typu „Hello World” włączający diodę LED na pełną jasność. Pozwoli ustawić środowisko pracy dla każdego z języków i zidentyfikować trudności z tym związane.
2. Regulacja obrotów silnika sterowanego sygnałem PWM generowanym sprzętowo za pomocą potencjometru.
3. Programowy regulator PID procesu dostosowywania częstotliwości obrotu jednego silnika do drugiego. Wartość zadana będzie przekazywana przez protokół HTTP z komputera zewnętrznego, gdzie będzie również raportowany i wyświetlany przebieg w czasie stanu regulatora (odczyty z czujników, wartość zadana, sterowanie). Część oprogramowania na komputerze będzie wspólna dla każdego z testowanych języków i nie będzie poddawana testom.
4. Program wykonujący złożone obliczenia, np. przetwarzanie obrazów. Po- może porównać wydajność języków.

## Potrzebny sprzęt

Poniżej zgromadziłem sprzęt, który będzie potrzebny do realizacji pracy.

- Raspberry Pi Zero W 512MB RAM - WiFi + BT 4.1
- ESP32 WiFi + BT 4.2
- Silnik DC 5V x2
- Czujnik Halla TLE4935 x2
- Magnesy neodymowe 5 mm  $\phi$ , 1 mm g $\acute{l}$ . x2
- Płytki stykowe
- Przewody
- Oporniki
- Potencjometry
- Diody LED

## Odnosińniki

- ankieta popularności języków TIOBE Index (ogólnie, nie tylko embedded): <https://www.tiobe.com/tiobe-index/>
- Comparison of high-level programming languages efficiency in embedded systems: <https://ui.adsabs.harvard.edu/abs/2019SPIE11176E..61C/abstract>
- Modern C++ and Rust in embedded memory-constrained systems: <https://odr.chalmers.se/items/18325574-018c-440e-a5f9-3e758bd4051c>
- Rust for secure IoT applications : why C is getting rusty: <https://digitalcollection.zhaw.ch/handle/11475/25046>
- Evaluating Python, C++, JavaScript and Java Programming Languages Based on Software Complexity Calculator (Halstead Metrics): <https://iopscience.iop.org/article/10.1088/1757-899X/1076/1/012046/meta>
- Halstead complexity measures: [https://en.wikipedia.org/wiki/Halstead\\_complexity\\_measures](https://en.wikipedia.org/wiki/Halstead_complexity_measures)

Trudno się szuka artykułów naukowych na temat Zig, nie potrafię znaleźć żadnych porównań z C.