Emilian ŚWITALSKI, Krzysztof GÓRECKI

Uniwersytet Morski w Gdyni, Wydział Elektryczny, Katedra Elektroniki Morskiej
ORCID: 1. 0000-0002-5437-381X 2. 0000-0002-9857-8235

doi:10.15199/48.2023.09.52

System zwalniania wątków VRTS jako alternatywa dla RTOS

***Streszczenie**: Artykuł opisuje implementację systemu zwalniania wątków **VRTS**, dostępnego na platformie GitHub pod adresem https://github.com/Xaeian/VRTS. Jest to alternatywa dla systemów czasu rzeczywistego RTOS w systemach wbudowanych. Przedstawiono różne koncepcje programowania mikrokontrolerów, zwracając uwagę na różnice między nimi. Omówiono funkcjonalności biblioteki VRTS, przedstawiając jej wykorzystanie w układzie zasilającym elektrolizer alkaliczny.*

***Abstract**: The article presents VRTS - cooperative multitasking, searchable on GitHub at https://github.com/Xaeian/VRTS. It is an alternative to real-time operating systems (RTOS) in embedded systems. Various concepts of programming microcontrollers were presented, paying attention to the differences between them. The functionalities of the VRTS library were discussed, presenting its use in the supply system of alkaline electrolyzer. (**Cooperative multitasking system VRTS as an alternative to RTOS**).*

Słowa kluczowe: Systemy wbudowane, RTOS, Zwalnianie wątków, Przełączanie wątków, Ansi C, Cortex, ARM
Keywords: Embedded systems, RTOS, cooperative multitasking, Thread switching, Ansi C, Cortex, ARM

Wstęp

Systemy wbudowane odgrywają ważną rolę na rynku elektroniki przemysłowej. W niektórych przypadkach komputer lub sterownik PLC nie są optymalnym, a nawet wystarczającym rozwiązaniem. Szczególnie w przypadku urządzeń wymagających elastyczności lub niskiej ceny końcowej [1]. Projektowanie dedykowanych pod konkretne rozwiązania urządzeń opartych na mikrokontrolerach pozwala na spełnienie tych wymagań.

Niestety wadą tego podejścia jest długi cykl testowania, który znacząco wydłuża czas wytwarzania prototypowych urządzeń. Na szczęście, obecnie wiele firm oferuje usługi wytwarzania płytek drukowanych *(PCB)* oraz montażu, w akceptowalnym czasie i za relatywnie niską cenę, zarówno na rynku krajowym, jak i międzynarodowym. Nawet uruchomienie własnej linii produkcyjnej dla prototypów w firmie nie jest obecnie dużą przeszkodą [2].

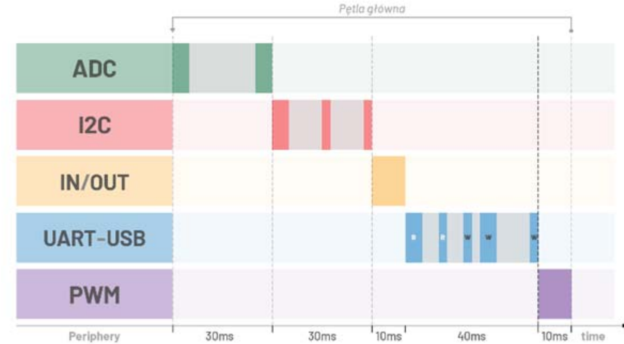
Kolejną wadą jest wysoki koszt prac deweloperskich związanych z pisaniem kodu, zarówno na etapie testowania, jak i wdrożenia [3]. Poza językiem programowania, istotny jest wybór metody tworzenia aplikacji, która umożliwi osiągnięcie określonego zestawu funkcjonalności, zachowując jednocześnie ograniczenia dotyczące zasobów sprzętowych *(pamięć i procesor)* w możliwie krótkim czasie.

W ramach tej pracy przedstawiono różne podejścia do tworzenia aplikacji systemów wbudowanych, wraz z omówieniem systemu zwalniania wątków, który stanowi istotę pracy. Następnie opisano autorską implementację zwalniania wątków **VRTS**, która została porównana z przykładem zastosowania w zasilaczu dedykowanym do elektrolizera. W podsumowaniu pracy porównano wszystkie przedstawione koncepcje, a także dokonano ich subiektywnej oceny.

Programowanie blokujące

Programowanie **sekwencyjne** *(blokujące)* polega na zatrzymywaniu wykonania kodu w momencie, gdy program napotyka na operację blokującą, taką jak oczekiwanie na wykonanie pomiaru przetwornikiem ADC. Podczas oczekiwania na zakończenie danej operacji, cały wątek jest zablokowany, a procesor pozostaje bezczynny *(rys. 1)*.

W systemach wbudowanych często dysponuje się niewielkimi mocami obliczeniowymi, a wykorzystanie ich efektywnie jest szczególnie ważne. Zatem istotne jest, żeby wykorzystywać metody programowania, które optymalizują **wydajność** systemu.



Rys 1. Linia czasu dla operacji wykonywanych sekwencyjnie

Co więcej, zwykle wiele zadań musi być wykonywanych jednocześnie, na przykład odbieranie i wysyłanie danych, przetwarzanie sygnałów, obsługa interfejsów, itp. W przypadku programowania sekwencyjnego każda operacja, która wymaga oczekiwania na dane, może spowodować opóźnienia w całym systemie, co może prowadzić do problemów z **reaktywnością**. Może to uczynić aplikację niefunkcjonalną w przypadku zbyt długich operacji blokujących.

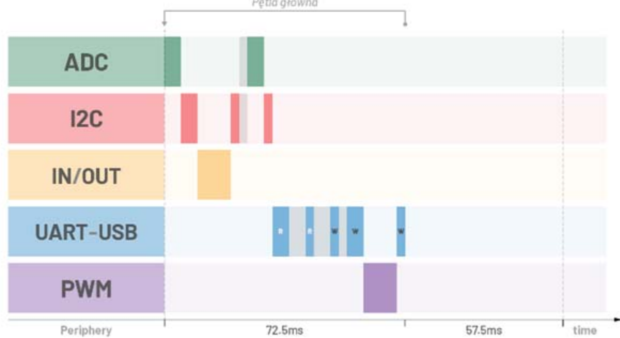
Dlatego też w systemach wbudowanych często stosuje się podejście **asynchroniczne**, w którym operacje blokujące są zastępowane przez operacje nieblokujące. Operacje takie nie zatrzymują wykonania kodu i pozwalają na równoczesne wykonywanie wielu zadań. W ten sposób procesor jest wykorzystywany w sposób efektywny, a system jest bardziej reaktywny i wydajny.

Programowanie reaktywne

Najprostszą techniką pracy asynchronicznej jest **polling**, czyli regularne sprawdzanie statusu wykonywanych zadań. Jednak w przypadku dużej liczby operacji, ta metoda staje się nieefektywna. Aby zwiększyć wydajność i reaktywność, można wykorzystać peryferia mikrokontrolera, takie jak liczniki *(timeout)*, DMA, przerwania itd., reagując jedynie na zdarzenia, bez ciągłego monitorowania stanu zadań. Taka metoda programowania, nazywana **reaktywną** lub **zdarzeniową**, rozwiązuje problem wydajności i reaktywności *(rys. 2)*.

Jednakże, wdrażanie takiego podejścia wymaga mieszania funkcjonalności na poziomie kodu lub dzielenia ich na małe, nieblokujące fragmenty. Może to zmniejszyć czytelność programu i utrudnić tworzenie bibliotek, co może

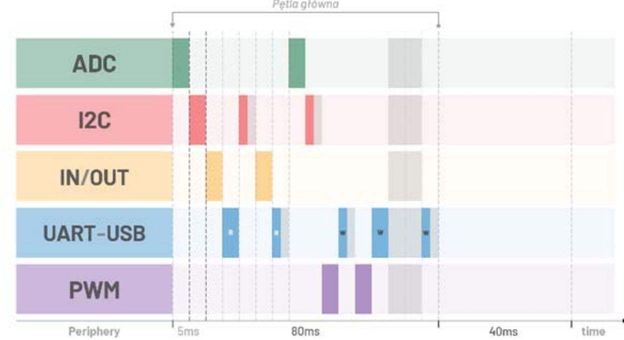
prowadzić do problemów z długoterminowym wsparciem aplikacji. Biblioteki stają się wówczas mocno związane z aplikacją, co uniemożliwia ich rozwój jako niezależnych modułów.



Rys 2. Linia czasu dla operacji wykonywanych reaktywnie

System czasu rzeczywistego RTOS

Rozwiązaniem jest mechanizm umożliwiający niezależny rozwój modułów przy równoległej *(współbieżnej)* obsłudze. Programowanie **wielowątkowe** stanowi skuteczną metodę, umożliwiającą realizację zadań w osobnych wątkach, a system *(biblioteka)* zajmuje się ich przełączaniem. Najczęściej stosowaną techniką przełączania wątków jest **system czasu rzeczywistego RTOS** [4] *(rys. 3)*. W najprostszej implementacji każdemu wątkowi przypisuje się równą porcję czasu na realizację zadań, zwykle wynoszącą `10ms`. Każdy wątek musi mieć również przydzieloną pamięć podręczną, co wymaga kopiowania danych między wątkami i zużywa zasoby procesora. Dlatego ważne jest, aby liczba wątków i czas przełączania były jak najmniejsze, jednocześnie zapewniając wymaganą deterministyczną czasowo reakcję na zdarzenia oraz szybkość działania systemu.



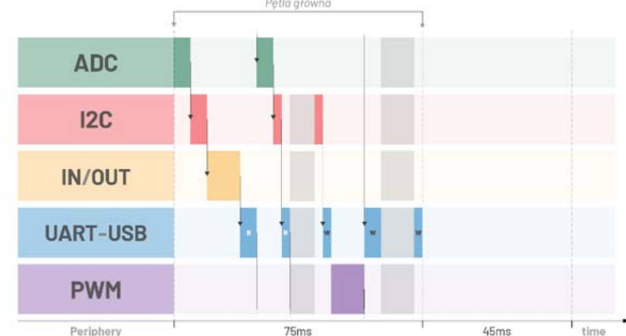
Rys 3. Linia czasu dla operacji wykonywanych w systemie czasu rzeczywistego

Niestety, stosowanie takiego podejścia do programowania może wiązać się z wieloma problemami związanymi ze współdzieleniem pamięci, takimi jak zakleszczenia czy wyścigi wątków. Aby rozwiązać te problemy, stosuje się mutexy, kolejkowanie lub inne mechanizmy zarządzające dostępem do pamięci, a w ostateczności może być konieczne wyłączenie systemu przerwań. Rozwiązaniem problemów związanych z pamięcią są **wzorzec aktorów** i **model CSP**, które zapewniają mechanizmy wymiany danych. Jednakże korzystanie z tych rozwiązań wiąże się z dodatkowym narzutem związanym z zużyciem procesora i pamięci, co może być nieakceptowalne dla wielu mniejszych aplikacji i projektów.

Niemniej, pisanie aplikacji wielowątkowych z wykorzystaniem opisanych metod różni się znacząco od aplikacji jednowątkowych, a same rozwiązania są stosunkowo skomplikowane i mogą wiązać się z dodatkowymi kosztami i trudnościami w procesie wdrożenia.

Zwalnianie wątków

Rozwiązaniem może być **zwalnianie wątków** *(cooperative multitasking)* [5]. W systemie tym każdy wątek dobrowolnie zwalnia sterowanie po wykonaniu swojego zadania, umożliwiając innym wątkom uzyskanie dostępu do procesora. W ten sposób wątki mogą się wzajemnie komunikować i koordynować bez konieczności stosowania złożonych mechanizmów wymiany danych i synchronizacji *(rys. 4)*.



Rys 4. Linia czasu dla operacji wykonywanych ze zwalnianiem wątków

System zwalniania wątków charakteryzuje się niskim narzutem czasowym i jest stosunkowo łatwy w implementacji i wdrożeniu, co czyni go atrakcyjną alternatywą dla RTOS w przypadku mniejszych projektów lub aplikacji wymagających niskiej latencji i krótkiego czasu reakcji.

System zwalniania wątków VRTS

Podstawowym założeniem autorskiej implementacji obsługującej zwalnianie wątków jest tworzenie i testowanie aplikacji w jednym wątku. Dzięki temu przeniesienie ich do środowiska wielowątkowego jest łatwe i ogranicza się do przekazania wątku i wywołania funkcji inicjującej system ich przełączania.

```
1 SYSTICK_Init(10); // 10ms
2 ROOT_Init(&root);
3 APP_Init();
4 thread(&Thread_1, stack_1, sizeof(stack_1));
5 thread(&Thread_2, stack_2, sizeof(stack_2));
6 VRTS_Init();
```

W tym celu biblioteka zapewnia kilka podstawowych funkcji, a najważniejszą z nich jest `let`. Po wywołaniu tej funkcji w aplikacji wielowątkowej następuje przełączenie wątku, natomiast w aplikacji jednowątkowej funkcja nie wykonuje żadnych działań. Sposób jej obsługi zależy od wartości definicji `VRTS_SWITCHING`.

```
 1 void UART_Thread(void)
 2 {
 3  while(1) {
 4    char *msg;
 5    size_t len = UART_Read(msg);
 6    if(len) {
 7    // message handling
 8    }
 9    let();
10  }
11 }
```

Oprócz obsługi przełączania wątków biblioteka obsługuje także licznik systemowy **SysTick**. Jest to niezbędne do obsługi zadań czasowych, które są ściśle powiązane z przełączaniem wątków. Aby móc w pełni wykorzystać ten licznik, biblioteka zapewnia także kilka dodatkowych funkcji.

Przykładami takich funkcji są **delay** oraz **sleep**, które pozwalają na oczekiwanie przez pewien określony czas przed przejściem do kolejnych instrukcji. Jednak funkcja **delay** pozwala innym wątkom na pracę w trakcie oczekiwania, w przeciwieństwie do funkcji **sleep**, która zmusza procesor do pozostania bezczynnym przez ten czas.

```
1 GPIO_Set(&gpio);
2 delay(1000); // 1s
3 GPIO_Rst(&gpio);
```

```
1 GPIO_Set(&gpio);
2 sleep(200); // 200ms
3 GPIO_Rst(&gpio);
```

Jedną z nieco mniej oczywistych, ale niezbędnych funkcji jest **timeout**, która określa maksymalny czas na wykonanie pewnego zadania. Wymaga ona przekazania funkcji **callback**, która zwraca flagę zwalniania **free**. W przypadku konieczności można również przekazać strukturę danych do funkcji. Jeśli zadanie nie zostanie wykonane w określonym czasie, funkcja zwróci wartość **true**, co umożliwia obsługę błędu. Oczywiście, podczas oczekiwania na realizację zadania, inne wątki mogą pracować.

```
1 ADC_Measurement(&adc); // start
2 uint8_t err = timeout(50, (void *)ADC_IsFree, &adc); // 50ms
3 if(err) {
4   // message handling
5 }
6 else {
7   uint16_t value = ADC_Get(&adc, 3); // channel 3
8   // do job
9 }
```

Istnieje również funkcja **waitfor**, która w połączeniu z **gettick** umożliwia opóźnienie obsługi pewnej operacji. Możliwe jest zaplanowanie i obsługa zadania w osobnych wątkach.

```
 1 uint64_t todo;
 2 void Thread_1(void)
 3 {
 4   if(!todo) todo = gettick(500); // 500ms
 5 }
 6 void Thread_2(void)
 7 {
 8    if(waitfor(&todo)) {
 9    // todo job
 9   }
10 }
```

Ostatnią funkcją jest **watch**, który pozwala na monitorowanie czasu od wywołania **gettick**. Dzięki niej możemy mierzyć czas wykonywania poszczególnych fragmentów kodu naszej aplikacji i identyfikować te, które są szczególnie czasochłonne.

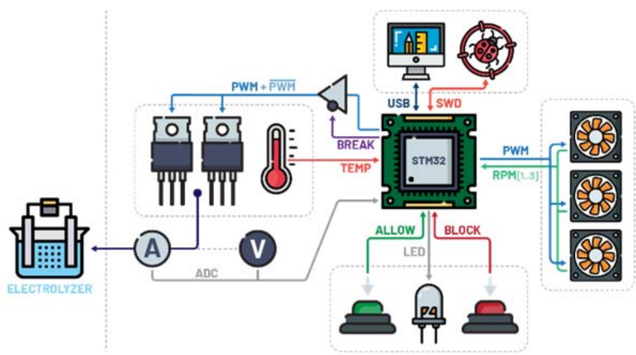
```
1 uint64_t stopwatch;
2 void Thread(void)
3 {
4   stopwatch = gettick(0); // no-offset
5   // calculations...
6    uint32_t time = watch(&stopwatch) // ms
7 }
```

Biblioteka została napisana w języku **Ansi C**. Pracuje z rdzeniami **Cortex** [6] **M4**, **M3**, **CM0+**. Jest dostępna na platformie **GitHub**: https://github.com/Xaeian/VRTS.

Implementacja

Autorski system **VRTS** został zaimplementowany w generatorze fali prostokątnej, zaprojektowanym specjalnie dla potrzeb badań elektrolizerów alkalicznych [7] *(rys. 5)*. Wykorzystany został w nim mikrokontroler **STM32G071RB** z rdzeniem **CM0+**. Jego charakterystyczną cechą jest wysoka wydajność prądowa *(do **120A**)* przy napięciu **12V** oraz liczne zabezpieczenia, które zwiększają jego niezawodność oraz zmniejszają szansę na uszkodzenia.

Zasilacz posiada możliwość generowania przebiegu prostokątnego o wypełnieniu i częstotliwości ustawianych przez użytkownika. W celu zapewnienia odpowiedniej pracy zasilacz mierzy temperaturę w pobliżu układów przełączających oraz steruje pracą wentylatorów chłodzących. W przypadku wystąpienia problemów, takich jak awaria wentylatorów, przekroczenie krytycznych wartości temperatury, prądu, napięcia szczytowego lub sygnałów bezpieczeństwa, zasilacz natychmiastowo przerywa zasilanie odbiornika w celu zapobieżenia uszkodzeniom.



Rys 5. Schemat blokowy zasilacza dla elektrolizera

Aplikacja została podzielona na pięć wątków:

- **MAIN**: obsługa interfejsu UART-USB do pracy z użytkownikiem oraz sterowanie wyjściem zasilania,
- **TEMP**: pomiar temperatury,
- **FAN**: sterowanie wentylatorami oraz nadzorowanie ich pracy,
- **ADC**: pomiar napięcia i prądu,
- **FUSE**: obsługa zabezpieczeń.

Podczas testowania, wątki zostały uruchomione oddzielnie, co umożliwiło swobodne debugowanie. Na koniec każdy z wątków został przekazany do kontrolera VRTS za pomocą funkcji **thread**.

Podsumowanie

Istnieje wiele różnych podejść do programowania. Nie jest możliwe jednoznaczne stwierdzenie, które jest bezwzględnie najlepsze. Decyzję należy podjąć na podstawie wymagań projektowych, takich jak koszt urządzenia, wydajność, wymagane funkcjonalności, energooszczędność, itp. nie zapominając o strategii rozwoju aplikacji oraz preferencji programistów. W celu ułatwienia tego procesu przedstawiono systematyzację podejść do programowania dla systemów wbudowanych *(tabela 1)* oraz poddano je subiektywnej ocenie pod względem:

- zużycia zasobów procesora i pamięci *(**mniejsza wartość** = mniejsze obciążenie procesora* i *mniejsze zużycie pamięci = **lepiej**)*,
- skalowalności i maksymalnego poziomu skomplikowania aplikacji *(**większa wartość** = można tworzyć bardziej złożone systemy = **lepiej**)*,

- popularności skorelowana z liczbą dostępnych materiałów i wielkością społeczności *(**większa wartość** = więcej gotowych rozwiązań, kursów = **lepiej**)*,
- trudności w opanowaniu i czytelności kodu *(**mniejsza wartość** = prostszy kod pisanej aplikacji o większej separacji części funkcjonalnych = **lepiej**)*.

System zwalniania wątków wydaje się ciekawą alternatywą dla RTOS, ponieważ jest on nieco prostszy i może być w stanie tworzyć rozwiązania *(systemy)* o podobnej złożoności przy nieco mniejszym zużyciu zasobów mikrokontrolera. Jednak obecnie nie jest to popularne rozwiązanie i jest ono polecane jedynie doświadczonym programistom, którzy są w stanie samodzielnie rozwiązywać napotkane przeszkody oraz dodawać funkcjonalności, które już zostały opracowane, np. w freeRTOS.

Tabela 1. Porównanie koncepcji programistycznych

	Metoda	Zużycie zasobów	Skalowalność	Popularność	Poziom trudności	Przykładowa implementacja
Programowanie jednowątkowe	sekwencyjne *(blokujące)*	1/8	1/4	3/4	1/3	ARDUINO
	polling *(odpytywanie)*	2/8	1/4	1/4	2/3	
	reaktywne *(zdarzeniowe)*	2/8	2/4	4/4	2/3	Cortex
Programowanie wielowątkowe	zwalnianie wątków	4/8	3/4	1/4	2/3	VRTS
	systemy czasu rzeczywistego	5/8	3/4	3/4	3/3	freeRTOS
	wzorzec aktorów, CSP	6/8	3/4	1/4	3/3	
Systemy operacyjne	mikrojądro	6/8	3/4	3/4	2/3	Zephyr
	monolityczne	8/8	4/4	4/4	1/3	
	wirtualizacyjne	8/8	4/4	2/4	1/3	

Przedstawione wyniki badań mogą być przydatne dla projektantów i programistów systemów wbudowanych, zwłaszcza dla tych, którzy korzystają z podejścia sekwencyjnego lub reaktywnego i napotykają trudności ze skalowaniem aplikacji. Dotyczy to również osób pracujących z systemami wbudowanymi, które rozważają optymalizację kosztową urządzeń poprzez wykorzystanie tańszych podzespołów, co wiąże się z dostosowaniem aplikacji do pracy z mniejszą ilością zasobów.

***Autorzy**: mgr inż. Emilian Świtalski, prof. dr hab. inż. Krzysztof Górecki, Uniwersytet Morski w Gdyni, Katedra Elektroniki Morskiej, ul. Morska 81-87, 81-225 Gdynia, E-mail: e.switalski@we.umg.edu.pl, k.gorecki@we.umg.edu.pl*

LITERATURA

1. A. Canedo, H. Ludwig and M. A. Al Faruque, "High Communication Throughput and Low Scan Cycle Time with Multi/Many-Core Programmable Logic Controllers," in *IEEE Embedded Systems Letters*, vol. 6, no. 2, pp. 21-24, June 2014, doi: 10.1109/LES.2014.2299731.
2. Atta-ul-Mustafa, S. Shakoor, S. Shoaib, W. Ahmed and H. Aleem, "Design and Development of Cost-effective PCB Prototyping Machine," *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*, St. Petersburg, Moscow, Russia, 2021, pp. 2006-2009, doi: 10.1109/ElConRus51938.2021.9396562.
3. M. Ullah, R. Ali, Abdullah, M. Ahmad, T. Khan and F. U. Mulk, "Software Cost Estimation – A Comparative Study of COCOMO-II and Bailey-Basili Models," *2019 International Conference on Advances in the Emerging Computing Technologies (AECT)*, Al Madinah Al Munawwarah, Saudi Arabia, 2020, pp. 1-5, doi: 10.1109/AECT47998.2020.9194166.
4. IEEE Standard for a Real-Time Operating System (RTOS) for Small-Scale Embedded Systems doi: 10.1109/AECT47998.2020.9194166.
5. J. Kopják and J. Kovács, "Timed cooperative multitask for tiny real-time embedded systems," *2012 IEEE 10th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Herl'any, Slovakia, 2012, pp. 377-382, doi: 10.1109/SAMI.2012.6208993.
6. ARM - Cortex-M0+ Devices, Generic User Guide https://www.keil.com/dd/docs/datashts/arm/cortex_m0p/r0p0/dui0662a_cortex_m0p_r0p0_dgug.pdf.
7. E. Świtalski, K. Górecki: System wbudowany do badania właściwości elektrolizera. Przegląd Elektrotechniczny, R. 98, nr 1, 2022, s. 151-154. doi: 10.15199/48.2022.01.31.