MINUT 2026 (8), s. 98–107
ISSN 2719-3063

Dariusz CABAN

Katedra Systemów Rozproszonych i Urządzeń Informatyki, Politechnika Śląska, ul. Akademicka 16, 44-100 Gliwice

Zastosowanie planisty zadań w programach dla mikrokontrolerów STM32

Streszczenie. W artykule przedstawiono sposób działania, funkcje oraz przykłady zastosowania niewywłaszczającego planisty zadań dla mikrokontrolerów STM32, który może być stosowany w oprogramowaniu systemu wbudowanego, któremu nie stawia się wygórowanych wymagań czasowych. Główną zaletą planisty jest eliminacja potrzeby używania znaczników wystąpienia zdarzeń, testowanych w głównej pętli programu. Jego zastosowanie zwiększa rozmiary wykonywalnego kodu oprogramowania oraz wykorzystywanej pamięci danych, ale wielkość wprowadzanych przez niego narzutów nie jest znaczna.

Słowa kluczowe: system wbudowany, mikrokontroler, zadanie, niewywłaszczający planista zadań.

1. Wstęp

Oprogramowanie systemu wbudowanego, któremu nie stawia się wygórowanych wymagań co do czasu odpowiedzi na zdarzenia zwykle zawiera program główny z pętlą nieskończoną, nazywaną też super pętlą, oraz podprogramy obsługi przerwań, jeśli wykorzystuje się je do sygnalizacji wystąpienia pewnych zdarzeń. W podprogramie obsługi przerwania ustawiany jest odpowiedni znacznik i w razie potrzeby są wykonywane najpilniejsze czynności związane z obsługą zdarzenia. W zależności od rodzaju zdarzenia cała jego obsługa lub mniej pilna część jest realizowana w pętli nieskończonej. Przy jednokrotnym sprawdzaniu w jednym obiegu pętli, czy zdarzenie wystąpiło, maksymalny czas odpowiedzi systemu będzie równy sumie czasu wykonania całego kodu w pętli i czasów wykonania podprogramów obsługi przerwań. Dostęp do danych współdzielonych między pętlę główną a podprogramy obsługi przerwań musi być odpowiednio zorganizowany, co może wiązać się z koniecznością zablokowania przerwań [3].

Wraz ze wzrostem liczby funkcji realizowanych przez oprogramowanie systemu kod umieszczony w pętli nieskończonej będzie się rozrastał, co może uczynić go trudnym do analizy. Rozwiązaniem tego problemu może być napisanie osobnych podprogramów obsługi zdarzeń, które będą stanowiły tzw. zadania, których wykonaniem będzie zarządzał niewywłaszczający planista CPU (ang. *CPU scheduler*), nazywany też kooperacyjnym [2], uruchamiany cyklicznie. Znaczniki wystąpienia zdarzeń będą niepotrzebne,

Autor korespondencyjny: D. Caban (Dariusz.Caban@polsl.pl).
Data wpłynięcia: 05.06.2026.

a konieczność wykonania zadania będzie zgłaszana planiście przy użyciu odpowiedniego podprogramu usługowego. Takie rozwiązanie zastosowała firma STMicroelectronics w oprogramowaniu np. urządzeń z interfejsem radiowym standardów Bluetooth Low Energy oraz IEEE 802.15.4, realizowanych w oparciu o mikrokontrolery serii STM32WB [4]. W materiałach firmowych planista jest określany słowem *sequencer* [5,6], tak jak planista opisany w książce [1]. W polskim wydaniu wspomnianej książki, które ukazało się w kwietniu 2026 roku, użyto słowa *sekwencer*.

2. Usługi planisty

Planista firmy STMicroelectronics został w całości napisany w języku C. Może on zarządzać wykonaniem do 32 zadań, zadanie stanowi funkcja, której nie przekazuje się argumentów i która nie zwraca wartości. Planista świadczy usługi:

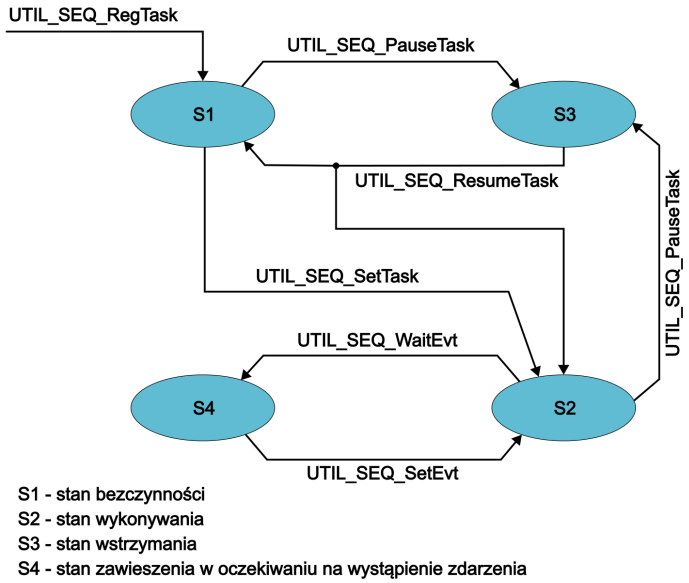
- utworzenia zadania,
- żądania wykonania zadania,
- wstrzymania/wznowienia zadania,
- zawieszenia zadania w oczekiwaniu na wystąpienie zdarzenia,
- sygnalizacji wystąpienia zdarzenia.

Zadaniu przy tworzeniu przypisuje się unikalny identyfikator. W żądaniu wykonania zadania określa się priorytet zadania, nie musi to być wartość unikalna. W sytuacji, kiedy nie ma zadań do wykonania lub zadań oczekujących na wystąpienie zdarzenia planista wywołuje funkcję, która może wprowadzić mikrokontroler w tryb oszczędzania energii.

W dowolnej chwili zadanie może być w jednym z kilku stanów:

- bezczynności,
- wykonywania,
- wstrzymania,
- oczekiwania na wystąpienie zdarzenia.

Na rysunku 1 przedstawiono możliwe przejścia między stanami. Przejścia, oprócz powrotu ze stanu wykonywania do stanu bezczynności, są wymuszane przy użyciu odpowiednich funkcji usługowych planisty, których nazwy są podane przy strzałkach.



Rysunek 1. Stany zadań i możliwe przejścia między nimi

3. Funkcje planisty

Poniżej opisano funkcje planisty dla mikrokontrolerów serii STM32WB oraz STM32WL, zawierających układ transmisji radiowej. W planiście dla mikrokontrolerów serii STM32C5, oficjalnie wydanym w połowie marca 2026 roku, inne są nazwy: funkcji, plików z kodem źródłowym oraz plików nagłówkowych [6].

Większości wymienionych dalej funkcji, w tym funkcji usługowych, jest przekazywany argument typu `UTIL_SEQ_bm_t`. Definicję tego typu zawiera plik `stm32_seq.h`:

```
typedef uint32_t UTIL_SEQ_bm_t
```

W przypadku funkcji `UTIL_SEQ_Run()` jest to maska bitowa do wskazania, które zadania mogą zostać wykonane – będą to te, w których identyfikatorach są wartości 1 na tych samych pozycjach co w masce. W przypadku pozostałych funkcji wartość tego typu jest identyfikatorem zadania lub zdarzenia, na którego wystąpienie zadanie ma oczekiwać. Tylko jeden bit identyfikatora powinien mieć wartość 1.

```
void UTIL_SEQ_Run(UTIL_SEQ_bm_t Mask_bm)
```

Funkcja ta musi być wywoływana w nieskończonej pętli w funkcji `main()` programu, z argumentem, którego wszystkie bity mają wartość 1 (np. ∼0). W jednym jej wywołaniu z taką maską uruchamiane są wszystkie zadania będące w stanie wykonywania. Odbywa się to w kolejności ich malejących priorytetów. Jeśli jest kilka zadań o tym samym priorytecie, to są one uruchamiane w kolejności ich rosnących wartości identyfikatorów.

```
void UTIL_SEQ_Init(void)
```

Inicjalizacja planisty – ustalane są wartości początkowe jego wewnętrznych zmiennych. Funkcja ta po-

winna być wywołana przed wejściem w nieskończoną pętlę programu.

`void UTIL_SEQ_Idle(void)`

Funkcja ta jest wywoływana przez planistę wtedy, gdy nie ma zadań do wykonania i zadań zawieszonych w oczekiwaniu na wystąpienie zdarzenia. Może w niej nastąpić wejście w tryb niskiego poboru energii.

Podane dalej funkcje są funkcjami usługowymi, służącymi do zmiany stanów zadań.

`void UTIL_SEQ_RegTask(UTIL_SEQ_bm_t TaskId_bm, uint32_t Flags, void (*Task)(void))`

Utworzenie zadania o podanym identyfikatorze `TaskId_bm`. Argument `Task` jest wskaźnikiem do funkcji, a wartość argumentu `Flags` nie jest wykorzystywana. Zadanie to znajdzie się w stanie bezczynności.

`void UTIL_SEQ_SetTask(UTIL_SEQ_bm_t TaskId_bm, uint32_t Task_Prio)`

Wprowadzenie zadania o identyfikatorze `TaskId_bm` i priorytecie `Task_Prio` w stan wykonania. Liczbę poziomów priorytetów zadań określa wartość stałej `UTIL_SEQ_CONF_PRIO_NBR`, zdefiniowanej w pliku `utilities_conf.h`. Domyślnie jest ona równa 2. Priorytet maleje wraz ze wzrostem swojej wartości.

`void UTIL_SEQ_PauseTask(UTIL_SEQ_bm_t TaskId_bm)`

Wprowadzenie zadania o podanym identyfikatorze w stan wstrzymania. Zadanie może wstrzymać samo siebie, ale po powrocie z funkcji `UTIL_SEQ_PauseTask()` kod tego zadania zostanie wykonany do końca. Funkcja ta może być wywoływana w podprogramach obsługi przerwań.

`void UTIL_SEQ_ResumeTask(UTIL_SEQ_bm_t TaskId_bm)`

Wprowadzenie zadania o podanym identyfikatorze w stan wykonywania lub bezczynności, w zależności od tego, w jakim stanie zadanie było przed wstrzymaniem. Funkcja może być wywoływana w podprogramach obsługi przerwań.

`void UTIL_SEQ_WaitEvt(UTIL_SEQ_bm_t EvtId_bm)`

Wprowadzenie zadania w stan oczekiwania na wystąpienie zdarzenia o podanym identyfikatorze. Powrót z tej funkcji następuje dopiero po zajściu tego zdarzenia, ale w tym czasie będą mogły być wykonywane pozostałe zadania. Jeżeli zawieszonych zostanie kilka zadań to będą one wznawiane w kolejności odwrotnej niż były zawieszane. Funkcji `UTIL_SEQ_WaitEvt()` nie należy wywoływać w podprogramach obsługi przerwań.

`void UTIL_SEQ_SetEvt(UTIL_SEQ_bm_t EvtId_bm)`

Powiadomienie planisty o wystąpieniu zdarzenia o podanym identyfikatorze. Funkcja ta może być wywoływana w podprogramach obsługi przerwań.

4. Przykłady użycia planisty

Moduł planisty składa się z kilku plików: `stm32_seq.c`, `stm32_seq.h` oraz `utilities_conf_template.h`. W pliku z kodem źródłowym znajdują się też, ujęte w dyrektywy kompilacji warunkowej, definicje stałych: `UTIL_SEQ_CONF_TASK_NBR` oraz `UTIL_SEQ_CONF_PRIO_NBR`, których wartości określają odpowiednio liczbę zadań (domyślnie 32) oraz liczbę poziomów priorytetu (domyślnie 2). Jeżeli użytkownik chce ustalić inne wartości tych parametrów to powinien do odpowiedniego katalogu swojego projektu skopiować trzeci z wymienionych plików, zmienić jego nazwę na `utilities_conf.h` i w nim zmienić wartości stałych o tych samych nazwach.

Do pliku `main.c` projektu utworzonego w środowisku STM32CubeIDE jest dołączany plik nagłówkowy `main.h`. Plik nagłówkowy należy w sekcji `Private includes` uzupełnić o dyrektywę dołączającą plik nagłówkowy `stm32_seq.h` i w razie potrzeby plik `utilities_conf.h`.

Planista jest wykorzystywany np. w programach z projektów przykładowych z biblioteki STM32CubeWB. Programy te są jednak stosunkowo złożone. W niniejszym artykule przedstawiono kilka prostszych przykładów użycia planisty [5]. Zamieszczone dalej programy zostały uruchomione i przetestowane na płytce rozwojowej P-Nucleo-WB55.

Przykład nr 1

Program, który co 400 ms zmienia stan zielonej diody LED na płytce P-Nucleo-WB55. Zmiana stanu diody jest realizowana w zadaniu `Task0`. Żądania jego wykonania są zgłaszane w funkcji obsługi przerwania od zegara systemowego `SysTick`, generującego sygnały żądania przerwania co 1 ms. Definicję stałej o wartości równej identyfikatorowi zadania należało zamieścić w pliku `main.h`, gdyż identyfikator musiał być znany funkcjom, których kod jest zamieszczony w różnych plikach źródłowych. Na listingach 1-3 przedstawiono uzupełnione fragmenty plików: `main.h` i `main.c` oraz uzupełniony kod źródłowy funkcji obsługi przerwania od zegara `SysTick`, który znajduje się w pliku `stm32wbxx_it.c`. Wymienione pliki zostały utworzone przez środowisko programistyczne. Występują w nich znaczniki w postaci komentarzy `/* USER CODE BEGIN ... */` i `/* USER CODE END ... */`, pomiędzy którymi należy umieścić swoje definicje, deklaracje oraz kod. Jeżeli nie będzie się przestrzegać tej zasady to wprowadzone uzupełnienia zostaną utracone przy rekonfiguracji projektu.

```
1 /* Exported constants ---------------------------------------- */
2 /* USER CODE BEGIN EC */
3 /* Identyfikator zadania */
4 #define TASK0  1 << 0
5 /* USER CODE END EC */
```

Listing 1. Definicja identyfikatora zadania w pliku nagłówkowym `main.h`

```
1 /* Private function prototypes ------------------------------- */
2 /* Prototypy innych funkcji */
3 /* USER CODE BEGIN PFP */
4 void function_Task0(void);
5 /* USER CODE END PFP */
6
7 int main(void)
```

```
8  {
9    /* MCU configuration ---------------------------------------- */
10
11
12   /* Infinite loop */
13   /* USER CODE BEGIN WHILE */
14   UTIL_SEQ_Init();
15   UTIL_SEQ_RegTask(TASK0, 0, function_Task0);
16
17   while (1)
18   {
19     /* USER CODE END WHILE */
20
21     /* USER CODE BEGIN 3 */
22     UTIL_SEQ_Run(~0);
23   }
24   /* USER CODE END 3 */
25 }
26
27 /* USER CODE BEGIN 4 */
28 void function_Task0(void)
29 {
30   HAL_GPIO_TogglePin(GPIOB, LED_GREEN);
31 }
32 /* USER CODE END 4 */
```

Listing 2. Fragmenty kodu programu głównego oraz kod zadania `Task0`

```
1  void SysTick_Handler(void)
2  {
3    /* USER CODE BEGIN SysTick_IRQn 0 */
4
5    /* USER CODE END SysTick_IRQn 0 */
6    HAL_IncTick();
7    /* USER CODE BEGIN SysTick_IRQn 1 */
8    if ((HAL_GetTick() % 400) == 0)
9    {
10     UTIL_SEQ_SetTask(TASK0, 0);
11   }
12   /* USER CODE END SysTick_IRQn 1 */
13 }
```

Listing 3. Kod funkcji obsługi przerwania od zegara systemowego

Przykład nr 2

Program, który miga zieloną diodą LED z różną częstotliwością w cyklach o czasie trwania 10 s. W

pierwszej połowie cyklu dioda miga z częstotliwością 4 Hz, a w drugiej z częstotliwością 0.5 Hz. W programie występują trzy zadania o różnych priorytetach: `TASK0` o najniższym, `TASK1` o pośrednim i `TASK2` o najwyższym. Wymagało to zmiany wartości stałej `UTIL_SEQ_CONF_PRIO_NBR`. Miganie diodą LED jest realizowane przez zadania: `TASK1` (z częstotliwością 0.5 Hz) i `TASK2` (z częstotliwością 4 Hz). W zadaniu `TASK0` są tylko zgłaszane żądania wykonania wszystkich zadań. Jest ono uruchamiane jako pierwsze, a żądanie jego wykonania jest zgłaszane tuż przed wejściem w pętlę nieskończoną. Potem wszystkie zadania są uruchamiane w kolejności ich malejących priorytetów. Użyte w programie stałe mogły być zdefiniowane w pliku `main.c`, zawartość jego odpowiednich fragmentów przedstawiono na listingu 4, sekcja z prototypami funkcji stanowiących zadania została pominięta.

```
1  /* Private define ------------------------------------------------------------*/
2  /* USER CODE BEGIN PD */
3  /* Identyfikatory zadan */
4  #define TASK0    1 << 0
5  #define TASK1    1 << 1
6  #define TASK2    1 << 2
7  /* Priorytety zadan */
8  #define PRIO_HIGH   0
9  #define PRIO_MEDIUM 1
10 #define PRIO_LOW    2
11 /* Parametry czasowe w ms */
12 #define TASK1_FREQ       1000/1
13 #define TASK2_FREQ       1000/8
14 #define TASK_TOGGLE_TIME 5000
15 /* USER CODE END PD */
16
17 int main(void)
18 {
19   /* MCU configuration --------------------------------------------------------*/
20
21
22   /* Infinite loop */
23   /* USER CODE BEGIN WHILE */
24   UTIL_SEQ_Init();
25   UTIL_SEQ_RegTask(TASK0, 0, function_Task0);
26   UTIL_SEQ_RegTask(TASK1, 0, function_Task1);
27   UTIL_SEQ_RegTask(TASK2, 0, function_Task2);
28   UTIL_SEQ_SetTask(TASK0, PRIO_LOW);
29
30   while (1)
31   {
32     /* USER CODE END WHILE */
33
34     /* USER CODE BEGIN 3 */
35     UTIL_SEQ_Run(~0);
```

```
36   }
37     /* USER CODE END 3 */
38 }
39
40 /* USER CODE BEGIN 4 */
41 void function_Task0(void)
42 {
43   UTIL_SEQ_SetTask(TASK0, PRIO_LOW);
44   UTIL_SEQ_SetTask(TASK1, PRIO_MEDIUM);
45   UTIL_SEQ_SetTask(TASK2, PRIO_HIGH);
46 }
47
48 void function_Task1(void)
49 {
50   uint32_t time_start = HAL_GetTick();
51   do {
52     HAL_GPIO_TogglePin(GPIOB, LED_GREEN);
53     HAL_Delay(TASK1_FREQ);
54   } while(HAL_GetTick() - time_start < TASK_TOGGLE_TIME);
55 }
56
57 void function_Task2(void)
58 {
59   uint32_t time_start = HAL_GetTick();
60   do {
61     HAL_GPIO_TogglePin(GPIOB, LED_GREEN);
62     HAL_Delay(TASK2_FREQ);
63   } while(HAL_GetTick() - time_start < TASK_TOGGLE_TIME);
64 }
65 /* USER CODE END 4 */
```

Listing 4. Definicje stałych oraz kod programu głównego i zadań dla przykładu nr 2

Przykład nr 3

Modyfikacja programu z przykładu nr 2 – w pierwszej połowie cyklu częstotliwość migania diody wynosi 0.5 Hz, a w drugiej 4 Hz. Zmianę częstotliwości migania uzyskano nie poprzez zmianę wartości stałych `TASK*_FREQ` lub zmianę priorytetów zadań, tylko przez wykorzystanie możliwości zawieszania zadań w oczekiwaniu na wystąpienie jakiegoś zdarzenia (listing 5). Najpierw jest uruchamiane zadanie `TASK0`, następnie zadanie `TASK2`, które od razu zawiesza się w oczekiwaniu na wystąpienie zdarzenia o identyfikatorze `EVENT0` (linia 34). Planista uruchamia zadanie `TASK1`, które przez 5 sekund realizuje miganie diodą z częstotliwością 0.5 Hz, po czym sygnaliizuje wystąpienie zdarzenia o identyfikatorze `EVENT0` (linia 28). Zadanie `TASK2` zostaje wznowione i przez kolejne 5 sekund dioda miga z częstotliwością 4 Hz. Przedstawiony ciąg kroków powtarza się w nieskończoność.

```
1 /* Private define ------------------------------------------------------------ */
2 /* USER CODE BEGIN PD */
```

```
3  /* Identyfikatory zadan */
4
5  /* Priorytety zadan */
6
7  /* Parametry czasowe w ms */
8
9  /* Identyfikator zdarzenia */
10 #define EVENT0   1 << 0
11 /* USER CODE END PD */
12
13 /* USER CODE BEGIN 4 */
14 void function_Task0(void)
15 {
16   UTIL_SEQ_SetTask(TASK0, PRIO_LOW);
17   UTIL_SEQ_SetTask(TASK1, PRIO_MEDIUM);
18   UTIL_SEQ_SetTask(TASK2, PRIO_HIGH);
19 }
20
21 void function_Task1(void)
22 {
23   uint32_t time_start = HAL_GetTick();
24   do {
25     HAL_GPIO_TogglePin(GPIOB, LED_GREEN);
26     HAL_Delay(TASK1_FREQ);
27   } while(HAL_GetTick() - time_start < TASK_TOGGLE_TIME);
28   UTIL_SEQ_SetEvt(EVENT0);
29 }
30
31 void function_Task2(void)
32 {
33   uint32_t time_start;
34   UTIL_SEQ_WaitEvt(EVENT0);
35   time_start = HAL_GetTick();
36   do {
37     HAL_GPIO_TogglePin(GPIOB, LED_GREEN);
38     HAL_Delay(TASK2_FREQ);
39   } while(HAL_GetTick() - time_start < TASK_TOGGLE_TIME);
40 }
41 /* USER CODE END 4 */
```

Listing 5. Definicja identyfikatora zdarzenia oraz kody zadań w programie wykorzystującym zawieszanie zadań w oczekiwaniu na wystąpienie zdarzenia

5. Zakończenie

Zastosowanie planisty zwiększy rozmiar wykonywalnego kodu całego programu systemu wbudowanego oraz wykorzystywanej pamięci danych. W materiałach firmowych brak danych o wielkościach narzutów pamięciowych wprowadzanych przez planistę. Dla ich oszacowania przygotowano prosty projekt dla zestawu P-Nucleo-WB55. Program źródłowy najpierw nie zawierał instrukcji wywołań opisanych wcześniej funkcji planisty, potem uzupełniono go o instrukcje wywołań wszystkich tych funkcji. Był on kompilowany w trybie Release z włączoną optymalizacją pod kątem rozmiaru kodu wykonywalnego. Kod wykonywalny programu w wersji z użyciem planisty zajął ok. 1 kB pamięci więcej. Z kolei zapotrzebowanie planisty na pamięć danych zależy od planowanej liczby zadań oraz poziomów ich priorytetu. Liczbę bajtów zajmowanych przez zmienne planisty można wyznaczyć z zależności $4 * n_1 + 8 * n_2 + 28$, gdzie n_1 – planowana liczba zadań, n_2 – liczba poziomów priorytetu.

Podziękowania

Autor pragnie podziękować recenzentom za trud włożony w recenzje.

Literatura

1. A. Mahmutbegović, S. Branam, *C++ in Embedded Systems. A practical transition from C to modern C++*, Packt Publishing, Birmingham 2025.
2. A. Silberschatz, P. B. Galvin, G. Gagne, *Podstawy systemów operacyjnych*, Wydawnictwo Naukowe PWN, Warszawa 2021.
3. D. E. Simon, *An embedded software primer*, Addison-Wesley Professional, 1999.
4. STMicroelectronics, *How to build wireless applications with STM32WB MCUs*, https://www.st.com/resource/en/application_note/an5289-how-to-build-wireless-applications-with-stm32wb-mcus-stmicroelectronics.pdf, 2024.
5. STMicroelectronics, *Sequencer*, https://wiki.st.com/stm32mcu/wiki/Utility:Sequencer, 2025.
6. STMicroelectronics, *Sequencer*, https://dev.st.com/stm32cube-docs/utilities-sequencer/2.0.0/en/docs/_tocs/sequencer_toc.html, 2026.