EMBEDDED OS
PROSTY SYSTEM OPERACYJNY PRZEZNACZONY NA PLATFORMY AVR

Autor: Bartłomiej Flak
Akademia Górniczo-Hutnicza

Spis treści

Lista oznaczeń

API	Application Programming Interface
CPU	Central Processing Unit
AVR	8-bit micro controller
OS	Operating System
ISP	In-System Programming
STL	Standard Template Library

1. Wstęp

Dokument dotyczy opracowania prostego systemu operacyjnego przeznaczonego do tworzenia aplikacji wbudowanych na mikrokontrolery z rodziny AVR. Działanie systemu ma polegać na cyklicznym przełączaniu wątków (procesów funkcyjnych definiowanych i dodawanych do kolejki zadań przez użytkownika) w pętli, sprawiając wrażenie równoległego wykonywania się procesów.

2. Wymagania systemowe (requirements)

Podstawowe założenia projektu:

1. Przygotowanie syntetycznego opisu architektury systemu.
2. Zdefiniowanie i zamodelowanie w UML wszystkich niezbędnych komponentów systemowych.
3. Opracowanie implementacji poszczególnych warstw aplikacji.
4. Implementacja zamodelowanego systemu na rzeczywistym urządzeniu.
5. Przeprowadzenie testów funkcjonalnych.

3. Funkcjonalność (*functionality*)

Ponieważ projekt zdefiniowano jako *prosty system operacyjny*, tylko podstawowe funkcjonalności takiego sytemu należy brać pod uwagę podczas modelowania i implementacji. Do tych funkcjonalności należą:

a) Przełączanie pomiędzy wątkami – głównym zadaniem systemu operacyjnego jest szybkie przełączane się pomiędzy różnymi wątkami w taki sposób, aby stworzyć wrażenie równoległego ich wykonywania. Zadania wykonywane są cyklicznie, zgodnie z przydzielonymi priorytetami.

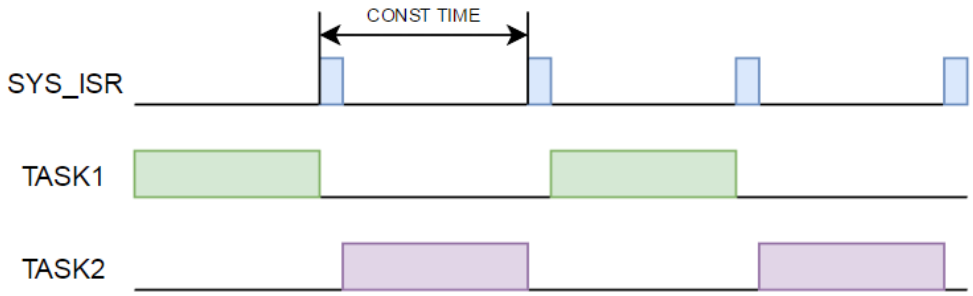
b) Możliwość podłączenia procesów funkcyjnych – system powinien pozwalać użytkownikowi na dodanie zdefiniowanych przez niego funkcji jako osobnych wątków systemowych. Wątki powinny być tworzone i usuwane w określony sposób, zdefiniowany przez API projektowanego systemu. To na użytkowniku spoczywa obowiązek zadbania o usuwanie zakończonych wątków przy użyciu odpowiednich funkcji.

c) Mechanizmy wykluczania (ang. *mutex*) – ponieważ wątki mogą współdzielić pewne obszary pamięci, należy uniemożliwić im jednoczesny dostęp do wspólnych zmiennych. Sytuacje te są groźne z punktu widzenia stabilności systemu, dlatego pożądanym rozwiązaniem jest wprowadzenie mutexów, czyli mechanizmów wzajemnego wykluczania, których celem jest zabezpieczenie wskazanych obszarów pamięci przed jednoczesnym dostępem kilku procesów.

d) Sekcje krytyczne – przez sekcje krytyczne należy rozumieć fragmenty kodu, które muszą być wykonywane bez przerwania. Stąd też system operacyjny musi pozwalać użytkownikowi na chwilowe wyłączenie przerwań. Użytkownik musi jednak zapewnić możliwe krótki czas wykonywania takiego *atomicznego* segmentu kodu, aby nie blokować pracy całego systemu.

4. Analiza problemu (*problem analysis*)

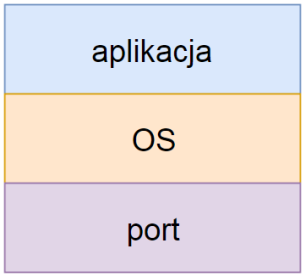
Głównym zadaniem systemu operacyjnego jest odpowiednie rozdzielenie zasobów platformy sprzętowej pomiędzy różne procesy. Przez zasoby sprzętowe należy rozumieć zarówno dostępną pamięć operacyjną jak i czas wykonywania (dostępność procesora) poszczególnych zadań. Jak wiadomo większość mikrokontrolerów (w tym mikrokontrolery z rodziny AVR) to urządzenia jedno wątkowe, zadaniem systemu operacyjnego jest stworzenie wrażenia wielowątkowości poprzez cykliczne przełączanie się pomiędzy różnymi procesami użytkownika:



Rysunek 4-1. Przebieg czasowy obrazujący aktualnie wykonywany proces.

W systemach z wywłaszczaniem poszczególne procesy mogą być przerywane na rzecz innych co sprawia wrażenie wielowątkowości. Koordynację tego procesu zapewnia planista (ang. *scheduler*). Oczywiście aby umożliwić realizację tego zadania niezbędne jest wprowadzenie przerwania systemowego (*SYS_ISR*), wewnątrz którego swoja pracę przeprowadza planista. Wykonuję on przełączenie kontekstu pomiędzy zadaniami znajdującymi się w kolejce zadań. W przypadku, gdy żaden proces nie jest dostępny (procesy użytkownika zostały wstrzymane lub zakończone), planista powinien zlecić wykonanie tzw. *idle task* – wątku systemowego, który nie wykonuję żadnego zadania, a jedynie pozwala na utrzymanie ciągłości działania systemu operacyjnego.

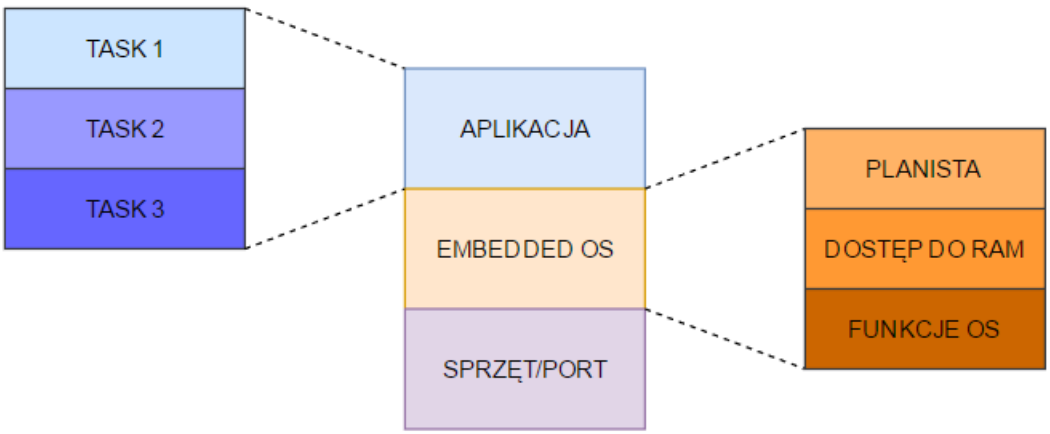
Strukturę aplikacji wbudowanej wykorzystującą do działania system operacyjny, możemy podzielić na 3 poziomy:



Rysunek 4-2. Struktura budowy aplikacji wbudowanej wykorzystującej system operacyjny.

Najwyższy poziom to warstwa aplikacji użytkownika. Ten segment przeznaczony jest dla osoby tworzącej funkcjonalność aplikacji, czyli ogół zadań/wątków oraz związany z nimi podział pamięci operacyjnej. Użytkownik taki powinien mieć zapewniony dostęp do uniwersalnego API systemu operacyjnego (warstwy środkowej). Jest to część kodu niezależna w żaden sposób od architektury sprzętu na jakim uruchamiano aplikację użytkową. Dobry system operacyjny powinien posiadać możliwość przeniesienia go na dowolną platformę. Aby to zrealizować należy wyszczególnić warstwę *portu*, czyli zbioru implementacji funkcji systemu operacyjnego, które bezpośrednio zależą od własności platformy uruchomieniowej. W tym wypadku będą to min. konfiguracja licznika wywołującego przerwania systemowe, implementacja mechanizmu przydzielającego wątkom pamięć RAM czy funkcję zapisu/odczytu rejestrów kontrolnych.

Kontynuując rozważania na temat architektury systemowej możemy wyszczególnić komponenty właściwych warstw struktury aplikacji wbudowanej, które przedstawiono na poniższej ilustracji:



Rysunek 4-3. Wyróżnione komponenty poszczególnych warstw architektury aplikacji wbudowanej.

Aplikacja użytkownika składa się z zadań (ang. *task*), które dodawane są do kolejki. Zadania realizują konkretne funkcjonalności, niemniej powinny działać one w pętli nieskończonej, gdyż nagłe zakończenie wątku, następujące przed wywołaniem przerwania systemowego i uruchomieniem do działania planisty, może spowodować przejście programu w nieokreślone miejsce, co oczywiście jest niepożądane. Dlatego też system operacyjny powinien udostępniać funkcję, wywoływane przez użytkownika w momencie zakończenia działalności przez dany wątek, których zadaniem będzie płynne przełączenie między zadaniami bez konieczności czekania na przerwanie systemowe.

Warstwa uniwersalna (środkowa) aplikacji wbudowanej składa się z planisty. Jego zadania zostały omówione wcześniej. Dodatkowo wyróżnić należy tutaj część programu odpowiedzialną za kontrolę zasobów pamięci (zmiennych) wykorzystywaną przez programy użytkownika. Są to mechanizmy takie jak *semafory* i *mutexy*. Ich zadaniem jest ograniczenie dostępności do danej przestrzeni pamięci jeżeli jeden z procesów aktualnie z niej korzysta. W projekcie z założenia zrezygnowano z tworzenia mechanizmów wzajemnego wykluczania (*mutex*), zdecydowano się jedynie na wykorzystanie prostych semaforów (wartość 0 lub 1) dostępowych.

Pod pojęciem *Funkcje OS* należy uwzględniać metody pozwalające na wprowadzenie czynnego oczekiwania wątku (zwykła funkcja *delay* jest tutaj nieadekwatna ze względu na marnowanie czasu procesora, wątek nie zostaje przerwany, wykonuję się dalej, czas marnowany w funkcji *delay*). Dzięki temu proces użytkownika, który aktualnie musi odczekać jakiś okres czasu, może zostać przerwany (wstrzymany na ten czas), a zamiast niego wykonywać się będzie inny wątek, co skutkuję lepszym, efektywniejszym wykorzystaniem czasu procesora CPU.

Oczywiście OS powinien udostępnią także funkcję zarządzające wątkami. Wśród nich należy bezwzględnie zdefiniować metodę tworzącą nowy wątek. Jej deklarację może wyglądać następująco:

```
TaskControlBlock* eCreateTask(
        eTaskHandler eTask,
        uint8_t eTaskStackSize,
        uint8_t eTaskPriority
);
```

Listing 4-2. Deklaracja funkcji tworzącej nowy wątek systemowy.

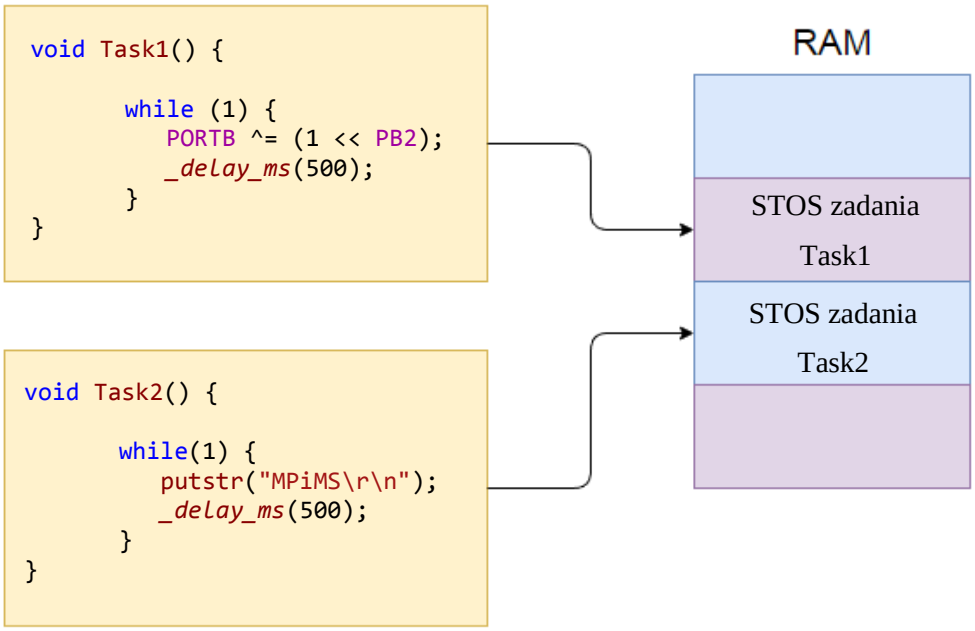
Funkcja jako parametry przyjmuję uchwyt do wskazanego zadania (*eTaskHandler* należy traktować jako wskaźnik na funkcję), ośmiobitowy rozmiar stosu oraz ośmiobitowy priorytet. Przykładowa funkcja spełniająca wymagania:

```
void task1(void) {

        while(1) {
           . . .
        }
        eTaskDestroy(NULL);
}
```

Listing 4-2. Przykładowa funkcja zadaniowa.

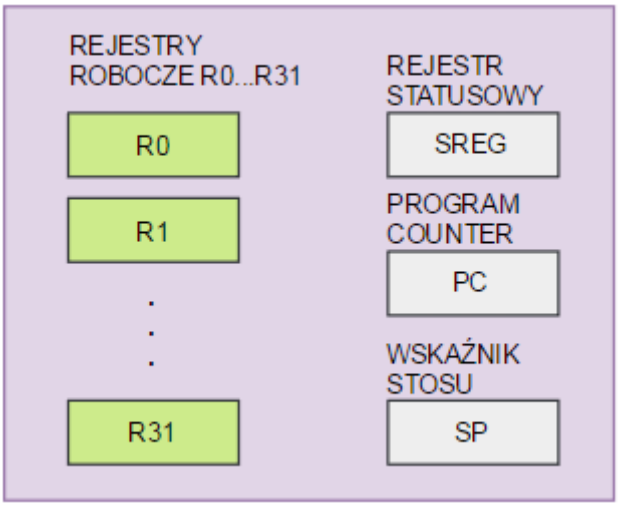
Wątek jest dodawany do kolejki systemowej w momencie wywołania funkcji *eCreateTask*. Zadaniem funkcji jest także przydzielenie odpowiednio dużej (podanej poprzez parametr *eTaskStackSize*) przestrzeni w pamięci przeznaczonej na indywidualny stos zadania.



Rysunek 4-4. Przydział pamięci RAM na lokalne stosy zadań.

Warstwa portu realizuję część systemu odpowiedzialną za przełączanie kontekstu pomiędzy wątkami. Jest to część kodu zależna od platformy sprzętowej, na której uruchamiana jest aplikacja. Niezależnie od systemów idea przełączania wątku skupia się na zapisaniu na stosie odpowiednich rejestrów kontrolnych wątku A, a następnie przywrócenie wartości rejestrów

wątku B i odpowiednie przesunięcie wskaźnika stosu. W przypadku rodziny AVR do dyspozycji mamy 32 rejestry ogólnego przeznaczenia (numerowane od R0 do R31), rejestr stanu (zawiera w sobie informację m.in. o przepełnieniu, a także poszczególne stany flag, rejestr licznika programu (przechowuję adres aktualnie wykonywanej instrukcji programu), a także wskaźnik stosu. Możemy zatem wydzielić pewien blok rejestrów, dzięki którym możliwe jest przejście między wątkami w dowolnym momencie.



Rysunek 4-5. Blok rejestrów systemowych – konieczne stworzenie kopii przy przełączaniu zadań.

Proces przełączania kontekstu można zatem podzielić na kilka etapów:

a) Wykonuję się zadanie A,

b) Następuje wywołanie przerwania systemowego, na stosie zapisywany jest adres powrotu,

c) Następuje proces zapisu na stosie kontekstu CPU (wartości poszczególnych rejestrów systemowych) oraz wskaźnik stosu danego wątku,

d) Po zapisaniu stanu wątku do pracy przystępuje planista, który wybiera z kolejki następne zadanie (B), jakie ma zostać wykonane,

e) Następuje odtworzenie wskaźnika stosu wątku B,

f) Odtwarzane są poszczególne rejestry systemowe wartościami zachowanymi na stosie wątku B,

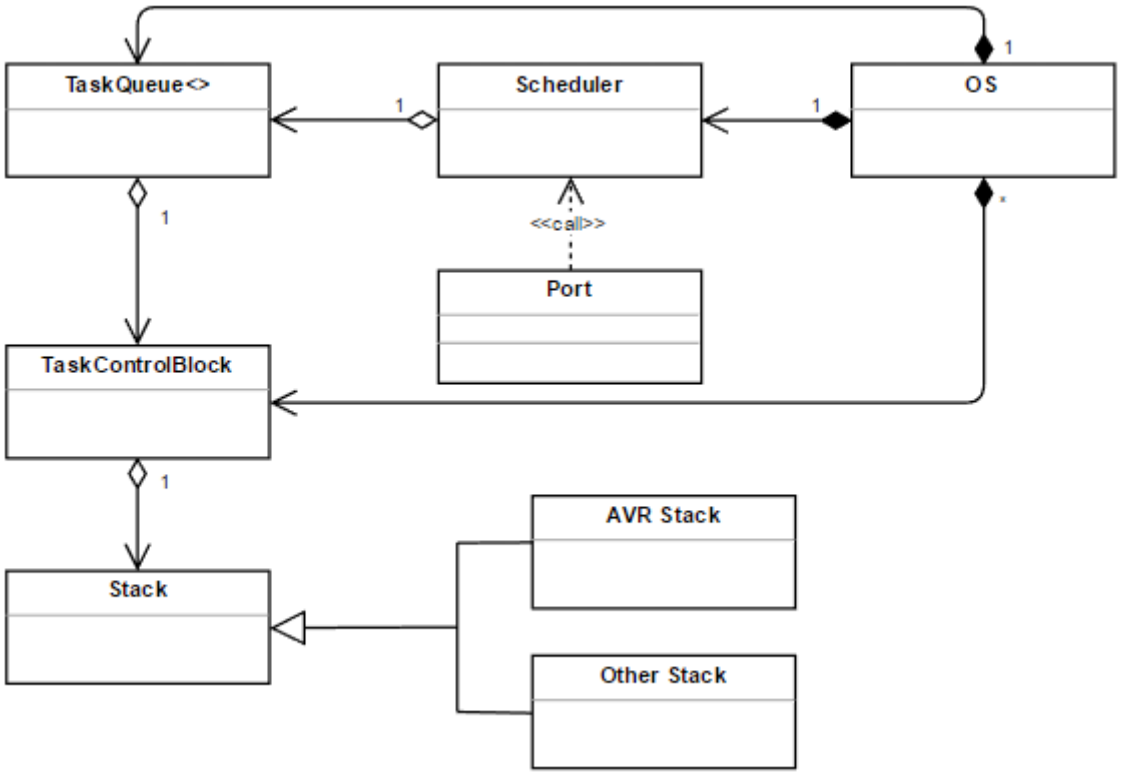
g) Następuje opuszczenie przerwania systemowego, program wraca do punktu wskazywanego przez wskaźnik stosu zadania B.

5. Projekt techniczny (*technical design*)

Implementacja system operacyjnego ma zostać dokonana przy użyciu języka C/C++, dlatego też wszystkie diagramy, wykresy, rysunki zawierają w sobie cechy charakterystyczne tego języka i zostały do niego przystosowane.

5.1.1 Hierarchia głównych klas projektu

System operacyjny składa się z kilku podstawowych komponentów. Hierarchia głównych klas została przedstawiona na Rys. 5.1.



Rysunek 5-1. *Hierarchia głównych klas projektu.*

Główną klasą projektu jest *OS*, to ona udostępnia metody, z których bezpośrednio może korzystać użytkownik aplikacji. Zawiera ona i zarządza klasą *Scheduler*, *TaskQueue* oraz *TaskControlBlock*. Ponadto klasa *OS* powinna być ograniczona tylko do jednej instancji (wzorzec: singleton). Dokładna definicja klasy *OS* znajduję się w 5.1.2

Scheduler zawiera w sobie odniesienie do kolejki oczekujących zadań, a także implementację przełączania kontekstu pomiędzy wątkami systemu.

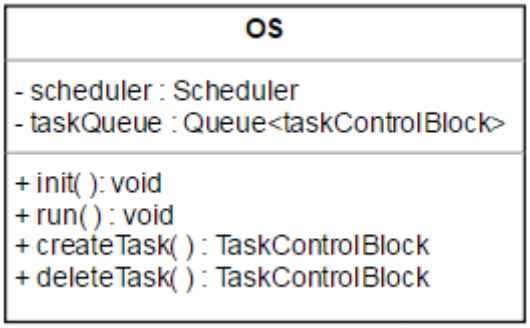
Template *TaskQueue* to klasa zawierająca implementację kolejki zadań. Jej elementy to bloki kontrolne wątków (*TaskControlBlock*) użytkownika.

Blok kontrolny wątku składa się z kilku pól, w tym z klasy *Stack*, która w zależności od architektury platformy sprzętowej, zawiera konkretną implementację metod bezpośrednio powiązanych z operowaniem na pamięci RAM mikrokontrolera (wzorzec: strategia).

Klasa *Port* związana jest zależnością z planistą, ponieważ zawiera w sobie wywołanie przerwania systemowego, które wyzwala jego działanie – proces odłożenia obecnego wątku na koniec kolejki i pobrania z niej następnego.

5.1.2 Opis klasy OS

Klasa *OS* jest główną klasą całego systemu operacyjnego. Dostarcza ona metod inicjalizujących system oraz go uruchamiających. Dodatkowo konieczne jest zaimplementowanie metod pozwalających na dodanie nowego wątku oraz jego usunięcie.



Rysunek 5-2. *Struktura klasy OS.*

Prywatne pola to instancja klasy *Scheduler* oraz klasa template *Queue* (kolejka zadań).

Opis metod:

init : metoda inicjująca działanie systemu, tworzy kolejkę zadań oraz inicjalizuję planistę

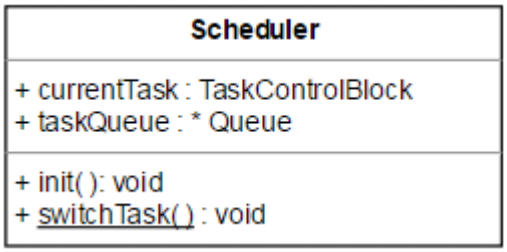
run : metoda uruchamiająca działanie systemu, po jej wywołaniu program powinien przejść do pierwszego zadania w kolejce i rozpocząć jego wykonywanie aż do momentu wystąpienia przerwania systemowego

createTask : metoda pozwalająca na dodanie funkcji użytkownika jako wątku systemowego, jako parametry przyjmę wskaźnik na ww. funkcję oraz parametry zadania (priorytet, rozmiar stosu)

deleteTask : metoda usuwająca wątek z kolejki zadań, dodatkowo zwalnia pamięć RAM

5.1.3 Opis klasy Scheduler

Klasa *Scheduler* odpowiedzialna jest za proces nadzory nad przełączaniem zadań. Implementacja powinna dostarczać algorytmy pozwalające wybrać następny proces uwzględniając jego priorytet oraz czas przebywania w kolejce.



Rysunek 5-3. *Struktura klasy Scheduler*

Opis pól:

currentTask : instancja obecnie wykonywanego zadania przez system

taskQueue : wskaźnik na kolejkę zadań (ustawiany na kolejkę w momencie i inicjalizacji systemu)

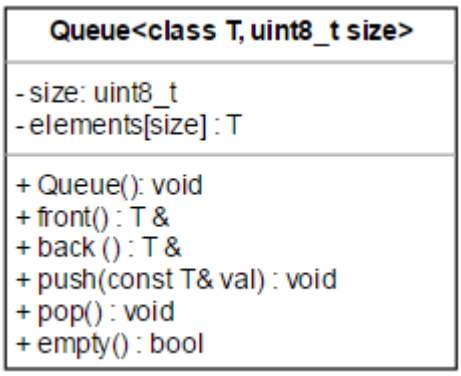
Opis metod:

init : metoda inicjalizująca planistę, w niej następuje ustawienie wskaźnika kolejki oraz aktualnego zadania przy inicjalizacji systemu operacyjnego

switchTask : metoda statyczna przełączająca zadania, wywoływana w przerwaniu systemowym (wewnątrz warstwy portu)

5.1.4 Opis klasy Queue<>

Klasa *Queue*<> implementuję kolejkę obiektów *T*, której rozmiar wynosi *size* (klasa ta do implementacji wykorzystuje dobrodziejstwa biblioteki STL).



Rysunek 5-4. *Struktura klasy Queue*<>

Opis pól:

s*ize* : rozmiar kolejki (liczba elementów)

elements[] : tablica o rozmiarze *size*, przechowująca elementy (*T*) kolejki

Opis metod:

Queue : domyślny konstruktor wywoływany przy tworzeniu instancji klasy, tworzy kolejkę o maksymalnym rozmiarze *size* oraz obiektach *T*

f*ront* : zwraca referencję do następnego elementu w kolejce („najstarszego" elementu)

back : zwraca referencję do ostatniego elementu kolejki – „najmłodszego" elementu

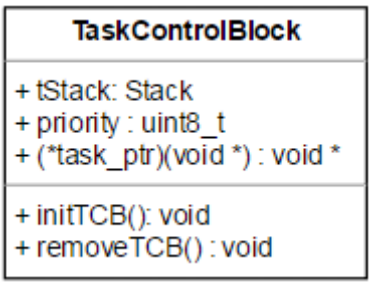
push : dodaje wskazany element *T* na koniec kolejki

pop : usuwa z kolejki „najstarszy" element – pierwszy w kolejce

empty : sprawdza czy kolejka jest pusta, następnie zwraca odpowiedni stan logiczny

5.1.5 Opis klasy TaskControlBlock

Klasa *TaskControlBlock* przechowuje wszystkie informację bezpośrednio związane z danym wątkiem. Są to zarówno odniesienia do struktury stosu jak i wskaźnik na funkcję zadaniową (definiowaną przez użytkownika systemu). *TaskControlBlock* to podstawowy element kolejki zadań, na której operuje planista.



Rysunek 5-5. *Struktura klasy TaskControlBlock*

Opis pól:

tStack : instancja klasy *Stack*, przechowująca informację dot. Lokalnego stosu zadania

priority : priorytet zadania (inwersja wartości)

task_ptr : wskaźnik do funkcji zadania o zdefiniowanym typie

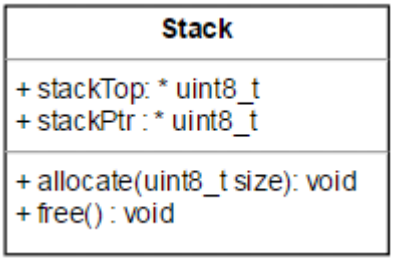
Opis metod:

initTCB : metoda inicjalizująca blok kontrolny (w tym uruchamia inicjalizację klasy *Stack*)

removerTCB : metoda usuwająca (zwalniająca) pamięć przeznaczoną na blok kontrolny (kasowanie bloku)

5.1.6 Opis klasy Stack

Klasa *Stack* jest implementowana indywidualnie w zależności od wykorzystywanej platform jednak jej interfejs pozostaje niezmieniony. Klasa ta ma na celu zapewnienie mechanizmów alokacji i zwalniania przestrzeni pamięci wykorzystywanej na potrzeby lokalnych stosów zadań. Klasa ta jest komponentem bloku kontrolnego zadania, definiując obszary pamięci przeznaczone na stos.



Rysunek 5-6. *Struktura klasy Stack.*

Opis pól:

stackTop : wskaźnik na adres początku lokalnego stosu zadania

stakPtr : wskaźnik lokalnego stosu zadania (*Stack Pointer*)

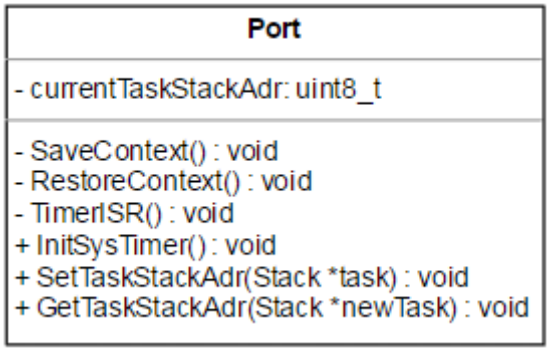
Opis metod:

allocate : rezerwuje miejsce w pamięci RAM przeznaczone na lokalny stos zadania, jednocześnie ustawiając wskaźniki *stackTop* oraz *stakPtr*

free : metoda zwalniająca zajmowane miejsce przez stos lokalny zadania

5.1.7 Opis warstwy Port

Port jest szczególnym komponentem całego system operacyjnego. Ta warstwa jest zależna bezpośrednio od sprzętu i powinna być podmieniana w zależności od używanej platformy (domyślnie AVR). Wyodrębnienie warstwy portu w znaczącym stopniu poprawi skalowalność całego systemu, czyniąc go łatwiej dostępnym dla różnych architektur.



Rysunek 5-7. *Struktura warstwy Port.*

Opis pól (zmiennych lokalnych) :

currentTaskStackAdr : adres początku stosu aktualnego zadania

Opis metod (funkcji) :

SaveContext : zapisuję wartości rejestrów systemowych na stos lokalny

RestoreContext : przywraca wartości rejestrów systemowych ze stosu lokalnego

TimerISR : obsługuję przerwanie systemowe – wywołanie *switchTask()* (patrz: 5.1.3)

InitSysTimer : inicjalizacja licznika systemowego (domyślnie przerwanie 60 Hz)

SetTaskStackAdr **:** ustawia *currentTaskStackAdr* na adres lokalnego stosu wskazanego zadania

GetTaskStackAdr : pobiera adres początku lokalnego stosu bieżącego zadania

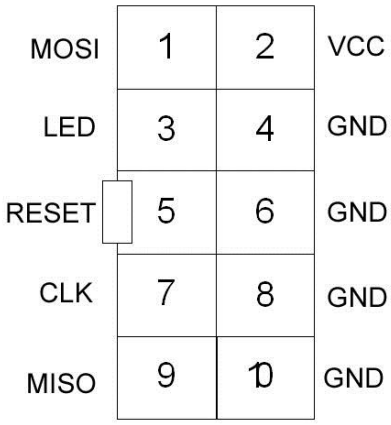
UWAGA:

Warstwa portu może wykorzystywać wstawki asemblerowe w celu dogodnego i bezproblemowego implementowania pewnych funkcjonalności.

6. Opis realizacji (*implementation report*)

6.1.1 Platforma sprzętowa

Implementacja projektu jest przeprowadzana przy wykorzystaniu mikrokontrolera ATmega16 w obudowie TQFP44. Do celów testowych użyto przygotowanej wcześniej platformy sprzętowej (płytki ewaluacyjnej) wyposażonej w ww. kontroler. Płytka ewaluacyjna zawiera w sobie interfejs USB-UART zrealizowany przy pomocy konwertera FT232RL oraz wyprowadzenia pinów wejścia/wyjścia (do wykorzystania w procesie debugowania). Programowanie mikrokontrolera możliwe jest dzięki wyprowadzonemu złączu ISP w standardzie CANDA:



Rysunek 6-1. Rozkład pinów złącza programatora.

6.1.2 Programator

Programowanie mikrokontrolera odbywa się za pomocą programatora *USBasp*. Jest to niekomercyjny projekt pochodzący ze strony http://www.fischl.de/usbasp/ . Projekt ten nie wspiera niestety procesu *debugowania*, ponadto nie jest wspierany jako natywne narzędzie w środowisku programistycznym Atmel Studio 7.0.

6.1.3 Zintegrowane środowisko programistyczne (IDE)

Do procesu implementacji zdecydowano się użyć dedykowanego środowiska programistycznego dostarczanego przez producenta mikrokontrolerów z rodziny AVR – Atmel Studio 7.0. Środowisko zbudowane jest na bazie Visual Studio i dostarcza wiele przydatnych

funkcjonalności, ponadto środowisko to dostarcza kompilator dedykowany dla mikrokontrolerów AVR – *AVR C Compiler*.

Głównym językiem programowania jest C++ w standardzie C11, jednak pewna część kodu (warstwa portu) wymagać będzie wstawek asemblerowych.

Projekt zorientowany jest w kierunku programowania obiektowego.

6.1.4 Tworzenie projektu w IDE

Implementację systemu operacyjnego rozpoczęto od stworzenia i przygotowania odpowiedniego projektu. Proces ten przebiega w następujących etapach:

a) *File -> New -> Project*

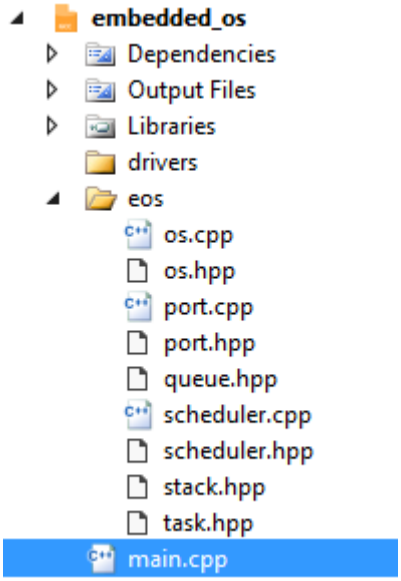
b) Ustawienia: *C/C++ / GCC C++ Executable Project*

c) Stworzono nowy folder w projekcie o nazwie *os*

d) *Project -> Properties*

e) W menu ustawień (*Properties*) dodano ścieżkę do utworzonego folderu oraz zdefiniowano zmienną globalną *F_CPU=14745000*

f) Utworzono pliki z rozszerzeniami *.cpp*/*.hpp* , których nazwy odpowiadają nazwą klas w diagramie klas (patrz: *Rys. 5.1*).



Rysunek 6-2. Struktura projektu.

6.1.5 Proces tworzenia kodu

Implementację systemu rozpoczęto od wypełnienia plików nagłówkowych odpowiednimi definicjami i deklaracjami uwzględnionymi w rozdz. 5.

W następnej fazie przystąpiono do oprogramowania konkretnych metod i funkcjonalności. Etap ten jest ciągle w fazie rozwoju i testowania.

7. Opis wykonanych testów (*testing report*)

Kod usterki	Data	Autor	Opis	Stan

8. Podręcznik użytkownika (*user's manual*)

8.1.1 Uruchomienie systemu

Listing X. przedstawia sposób na konfigurację i uruchomienie systemu operacyjnego przy pomocy dostępnego API. Aplikacja składa się z dwóch procesów, które są kolejno dodawane do kolejki zadań. Po zarejestrowaniu wątków następuję uruchomienie systemu.

```
#include <avr/io.h>
#include "util/delay.h"
#include "serial.hpp"
#include "os.hpp"

//Komponent portu szeregowego
Serial serialPort(9600);

//Funkcja zadaniowa numer 1
void myTask1(void) {
	while(1) {
		serialPort.putstr("World!\r\n");
		_delay_ms(200);
	}
}
//Funkcja zadaniowa numer 2
void myTask2(void) {
	while(1) {
		serialPort.putstr("Hello ");
		_delay_ms(100);
	}
}

int main(void) {

	//Tworzymy instancje systemu operacyjnego
	eOS os;
	//Inicjalizujemy system
	os.init();
	//Dodajemy pierwszy watek
	os.eCreateTask(task01, NULL, 64, 4);
	//Dodajemy drugi watek
	os.eCreateTask(task02, NULL, 64, 4);
	//Uruchamiamy system operacyjny
	os.run();

	//Program nigdy nie powinien dojsc do tego miejsca
	return 0;
}
```

Listing 8-1. Przykładowa aplikacja wykorzystująca Embedded OS.

UWAGA:

Aby uruchomić działanie systemu operacyjnego koniecznie jest załączenie odpowiednich bibliotek (patrz: *7.1.2*).

8.1.2 Konfiguracja projektu

//TODO

9. Metodologia rozwoju i utrzymania systemu (*system maintenance and deployment*)

//TODO

Bibliografia

[1] Gręboszˇ J.: Symfonia C++, Kraków 2000

[2] Mikrokontroler: wszystko o jego działaniu, http://forbot.pl/blog/artykuly/teoria/mikrokontroler-wszystko-co-powinniscie-wiedziec-o-jego-dzialaniu-id1314, 30.04.2017

[3] Atmel AVR Instruction Set, https://en.wikipedia.org/wiki/Atmel_AVR_instruction_set, 30.04.2017

[4] RTOS Task Switching: An Example Implementation In C, http://www.eetimes.com/document.asp?doc_id=1255034, 30.04.17

[5] Simple AVR RTOS – C version, https://teknoman117.wordpress.com/2012/06/13/simple-avr-rtos-c-version-2/, 30.04.2017

[6] RTOS implementation: project overview, http://web.uvic.ca/~rayhan/csc560/projects/project3/context_switch.html, 30.04.2017

[7] How to write small RTOS, https://larrylisky.com/2012/07/14/how-to-create-a-small-rtos/, 30.04.2017