KaRTOS

8–bitowe jądro czasu rzeczywistego, część 2

W pierwszej części cyklu została wstępnie poruszona teoretyczna problematyka systemów operacyjnych z uwzględnieniem systemów operacyjnych czasu rzeczywistego. Kontynuacją będzie przedstawienie prostej implementacji dla małych, tanich i powszechnie dostępnych, a jednocześnie wydajnych mikrokontrolerów jednoukładowych RISC. Zaprezentujemy również aplikację demonstracyjną.

Budowa systemu KaRTOS

System KaRTOS składa się z dwóch głównych części: podstawowej i rozszerzającej, co pokazano schematycznie na **rys. 7**. Część podstawowa jest niezbędna do pracy systemu. Jej uruchomienie jest konieczne. W skład części podstawowej wchodzą:

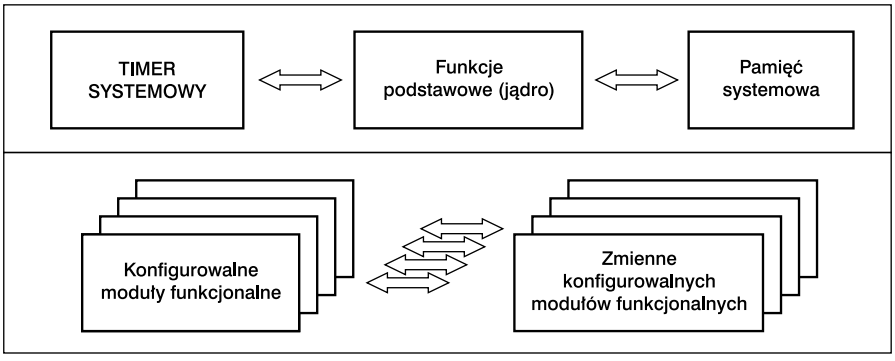
Timer systemowy – to blok wykorzystujący sprzętowy licznik mikrokontrolera i zegar systemowy do odmierzania jednostkowych odcinków czasu o długości 1 ms. Jest to minimalny kwant czasu, którym zarządza system KaRTOS.

Funkcje podstawowe (jądro) – to zbiór niezbędnych procedur realizujących takie zadania jak: inicjalizacja mikrokontrolera (również timera systemowego), inicjalizacja zadań, szeregowanie zadań, zarządzanie kolejkami.

Pamięć systemowa – to obszar pamięci danych zawierający zmienne systemowe oraz stos systemowy.

Część rozszerzająca systemu składa się z konfigurowalnych autonomicznych modułów rozszerzających funkcjonalność jądra. W jej skład wchodzą również wszystkie sterowniki układów peryferyjnych. W zależności od docelowej aplikacji, poszczególne podsystemy mogą być włączane lub wyłączane na etapie kompilacji kodu. Minimalna kompilacja systemu w wersji 3.00 z włączonym algorytmem karuzelowym szeregowania zadań zajmuje jedynie 1654 bajty pamięci ROM i 13 bajtów pamięci RAM.

W **tab. 1** pokazano zapotrzebowanie na pamięć programu ROM przez główne moduły systemu KaRTOS w wersji 3.00.



Rys. 7. Budowa systemu KaRTOS

Tab. 1. Zapotrzebowanie na pamięć programu ROM przez główne moduły systemu KaRTOS w wersji 3.00

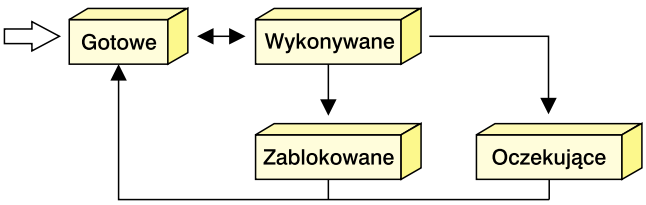
Lp.	Nazwa modułu	Rozmiar [bajty]	Opis
0.	KaRTOS wer. 3.01	1654	Podstawowa wersja systemu.
1.	KaRTOS_PRIORITY_SCHED	160	Włączenie algorytmu szeregowania round–robin z priorytetami i określeniem maksymalnego czasu dla zadania.
2.	KaRTOS_RTC	242	Moduł zegara czasu rzeczywistego działającego w oparciu o główny zegar systemowy.
3.	KaRTOS_32KHZ_RTC	294	Moduł zegara czasu rzeczywistego działającego w oparciu rezonator 32,768 kHz.
4.	KaRTOS_EXT_TIME	180	Uruchomienie rozszerzonej wersji zegara z datą.
5.	KaRTOS_UART_ON	1248	Sterownik wraz z funkcjami obsługi portu szeregowego.
6.	KaRTOS_ADC_ON	148	Sterownik wraz z funkcjami obsługi przetwornika ADC
7.	KaRTOS_EEPROM_ON	132	Sterownik wraz z funkcjami obsługi pamięci EEPROM.
8.	KaRTOS_STRING	206	Pakiet funkcji do manipulacji na ciągach znaków.
9.	KaRTOS_SEM	234	Moduł implementujący semafory.

Zadania w systemie KaRTOS

Każde zadanie uruchomione w systemie KaRTOS posiada swoje własne zasoby. Należą do nich:

Kod – instrukcje wykonywane przez kontroler znajdujące się w pamięci ROM. Wielkość zajmowanej pamięci zależy od algorytmu realizowanego przez zadanie. Zadanie mrugania diodą zajmie kilkanaście bajtów w pamięci programu, natomiast implementowanie skomplikowanych funkcjonalności wymaga większej jej pojemności.

Stos – obszar pamięci RAM zawierający adresy powrotu przy wykonywaniu skoków do funkcji, zmienne tymczasowe oraz kontekst zadania niezbędny podczas jego przełączania (rejestry danych, wskaźnik stosu, rejestr statusu kontrolera). Rozmiar tej struktury jest definiowany przez programistę implementującego algorytm zadania. Rozmiar stosu musi być tym większy, im więcej stopni zagnieżdżeń posiada korzystające z niego zadanie. Dla średnio rozbudowanego zadania



Rys. 8. Krążenie zadań w systemie KaRTOS

wystarczający będzie stos o wielkości 100 bajtów.

Blok kontrolny (*Task Control Block*) – obszar pamięci o rozmiarze 8 bajtów zawierający informacje o zadaniu. Umożliwia on identyfikację zadań i zarządzanie nimi przez system. TCB zawiera:

– numer identyfikacyjny zadania – PID (*Process ID*) – jest to liczba z zakresu od 2 do 255 jednoznacznie identyfikująca zadanie w systemie. PID numer 0 jest zarezerwowany, a numer PID o numerze 1 posiada systemowy proces bezczynności. Oczywistym jest, że w poprawnie działającym systemie nie mogą istnieć dwa zadania o identycznym numerze identyfikacyjnym. PID jest nadawany zadaniu podczas jego tworzenia,
– priorytet – liczba z zakresu od 1 do 254 określająca ważność zadania – im mniejsza wartość, tym zadanie jest ważniejsze. Najważniejsze jest zadanie z priorytetem 1, gdyż priorytet 0 jest zarezerwowany. Systemowy proces bezczynności posiada najmniejszą ważność (priorytet 255). Jeśli w systemie istnieją zadania o tym samym priorytecie oznacza to, że są równie ważne. W skrajnym przypadku wszystkie uruchomione zadania mogą posiadać identyczny priorytet i być traktowane na równi,
– licznik – jest zmienną o szerokości 16 bitów, przechowującą wartość (w ms) interwału czasu, na jaki zadanie zostało zawieszone,
– wskaźnik do struktury TCB umożliwiający realizację kolejek zadań,
– wskaźnik stosu przechowujący adres wierzchołka stosu zadania.

Każde zadanie w systemie może znajdować się w jednym z czterech stanów: wykonywane, gotowe, oczekujące lub zablokowane. Wędrówkę zadań w systemie KaRTOS pokazano na **rys. 8.**

Wystartowane zadanie znajduje się w stanie *gotowe* do momentu, aż zostanie uruchomione (wykonywane) zgodnie ze swoim priorytetem. Zadanie wykonywane to takie, które jest aktualnie w posiadaniu procesora. Oczekujące zadanie to takie, które przerwało swoje działanie na określony czas. W stanie *zablokowane* znajduje się zadanie, które oczekuje na określony zasób (dostęp do pamięci współdzielonej RAM, pamięci EEPROM, port itp.) lub na określony komunikat synchronizujący. Innymi słowy, w stanie *zablokowane* znajduje się zadanie nie mogące kontynuować działania z powodu innego niż oczekiwanie na odmierzenie interwału czasu.

Implementując algorytm zadania należy pamiętać, aby nie dopuścić do wyjścia z funkcji zadania. Można to zrealizować na różne sposoby:

– zadanie to nieskończona pętla

```
void Task_2(void)
{
    for(;;)
    {
        //Instrukcje zadania ;
    } ;
```

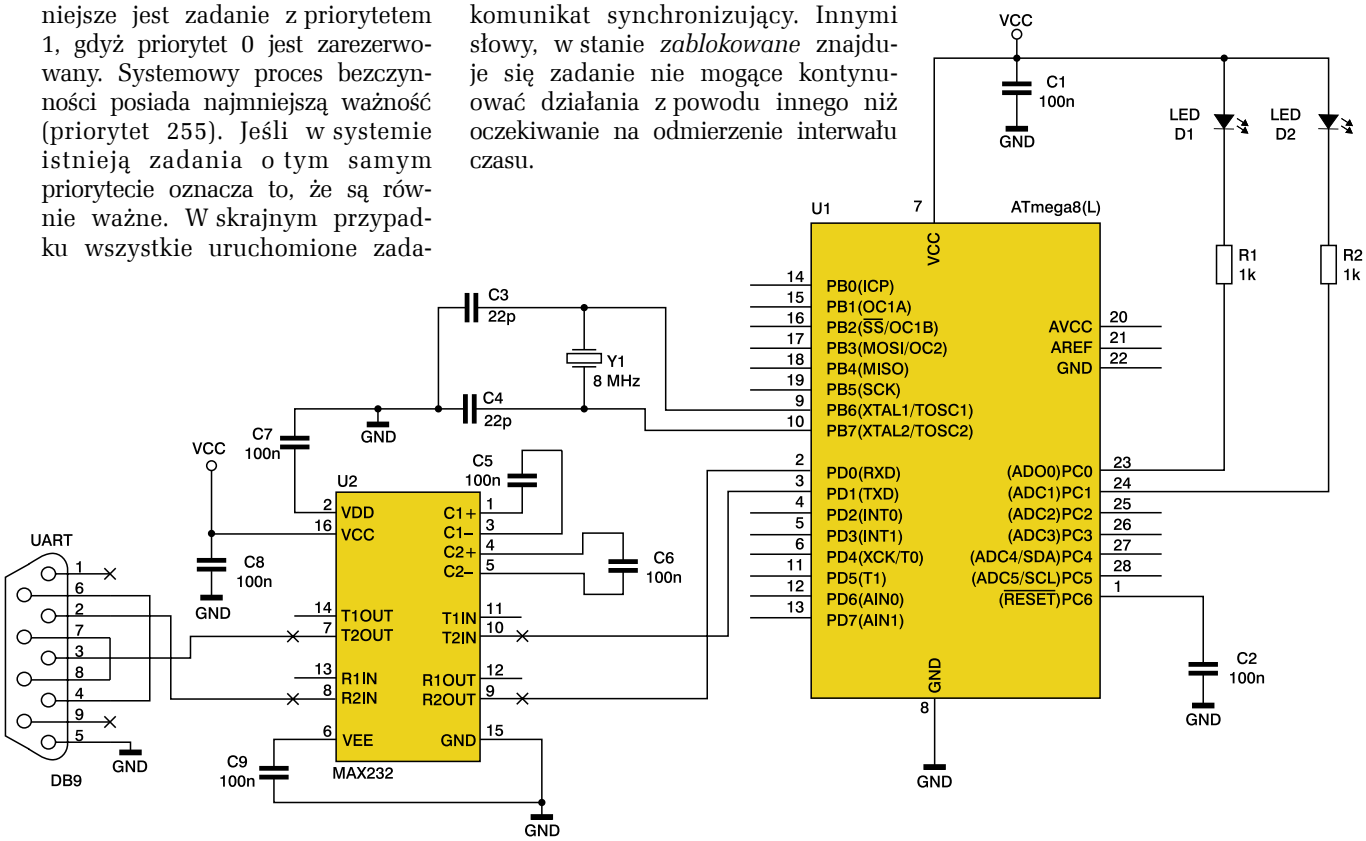
– nieskończona pętla znajduje się na końcu kodu, po zakończeniu algorytmu zadania

```
void Task_2(void)
{
    //Instrukcje zadania ;
    //Instrukcje zadania ;
    for(;;){ TimeSleepms(1000)
  ; }
}
```

„Hello wolrd tu KaRTOS", czyli pierwsza aplikacja w systemie KaRTOS

Po przedstawionym powyżej minimalnym wstępie teoretycznym nadszedł wreszcie czas upragniony przez wszystkich praktyków, czyli koniec marudzenia – przystępujemy do działania. Co jest potrzebne do tego, by napisać i uruchomić aplikację w systemie KaRTOS:

1. system KaRTOS – zawarty w pliku *Hello_world_tu_KaRTOS.zip* – jest to gotowa aplikacja demonstracyjna opisywana poniżej. Po rozpakowaniu pliku w katalogu Hello_world_



Rys. 9. Schemat układu dla aplikacji demonstracyjnej systemu KaRTOS

tu_KaRTOS znajdą się następujące zasoby:
- katalog KaRTOS – zawiera system operacyjny,
- plik *main.h* – zawiera parametry zadań (TASK_PID – numer zadania (przyjmuje wartości od 2 do 255), TASK_STACK – określa rozmiar stosu zadania w bajtach, TASK_PRIORITY – numer priorytetu zadania (przyjmuje wartości od 1 do 254)),
- plik *main.c* – tutaj zawarte są procedury tworzenia, inicjalizacji zadań oraz inicjalizacji i uruchomienia systemu operacyjnego,
- pliki *Task_1.h*, *Task_2.h*, *Task_3.h*, *Task_4.h* – pliki nagłówkowe zadań istniejących w systemie – zawierają deklaracje funkcji i zmiennych globalnych poszczególnych zadań,
- pliki *Task_1.c*, *Task_2.c*, *Task_3.c*, *Task_4.c* – pliki zawierające kod poszczególnych zadań,
- *makefile* – plik z instrukcjami automatycznej kompilacji dla programu make.

2. Kawałek sprzętu, czyli mikrokontroler ATmega8(L) wraz z peryferiami jak na **rys. 9** (jeśli do kontrolera jest podłączony rezonator o innej częstotliwości (lub nie jest podłączony żaden), zaprogramuj go do pracy z wewnętrznym oscylatorem o częstotliwości 8 MHz).
3. Zainstalowany kompilator avr–gcc najlepiej WinAVR 20040720,
4. Zainstalowany programator pozwalający załadować kod wynikowy do kontrolera ATmega8(L).

Posiadając powyższe składniki możemy przystąpić do dzieła, jakim jest napisania i uruchomienia pierwszej aplikacji dla systemu KaRTOS. Uruchomimy trzy zadania:
- Task_1 – będzie wysyłało co 1 sekundę przez port szeregowy ekran powitalny aplikacji – „Hello world tu KaRTOS !". Parametry transmisji: 8N1,38400 (osiem bitów danych, bez bitu parzystości, jeden bit stopu z prędkością 38,4 kbps),
- Task_2 – będzie zapalało na 1 sekundę i gasiło na 1 sekundę diodę_1 podłączoną do pinu PC0 mikrokontrolera,
- Task_3 – będzie zapalało na 100 ms i gasiło na 400 ms diodę_2 podłączoną do pinu PC1 mikrokontrolera.

Tab. 2. Zmienne zdefiniowane w systemie KaRTOS

Typ zmiennej w systemie	Opis zmiennej
u08	zmienna 8–bitowa bez znaku
s08	zmienna 8–bitowa ze znakiem
u16	zmienna 16–bitowa bez znaku
s16	zmienna 16–bitowa ze znakiem
u32	zmienna 32–bitowa bez znaku
s32	zmienna 32–bitowa ze znakiem

List. 1. Parametry zadań w pliku *main.h*
```
#define TASK_1_PID              10
#define TASK_1_STACK            100
#define TASK_1_PRIORITY         10
#define TASK_2_PID              20
#define TASK_2_STACK            100
#define TASK_2_PRIORITY         20
#define TASK_2_PID              30
#define TASK_2_STACK            100
#define TASK_2_PRIORITY         30
```

Zmienne

Zmienne zdefiniowane w systemie KaRTOS przedstawiono w **tab. 2.**

Konfiguracja zadań

Otwórz plik *main.h* i upewnij się, że parametry zadań TASK_1, TASK_2 i TASK_3 są odpowiednio skonfigurowane (**list. 1**).

Uruchomienie zadań

Otwórz plik *main.c* i upewnij się, że zadania Task_1, Task_2 i Task_3 zostaną utworzone i uruchomione (**list. 2**). Funkcja *void KaRTOSTaskInit(void(*pMyfunction)(void), u08 Pid, u08 u08Prio, u16 u16StackSize)* przyjmuje następujące parametry:
- `void(*pMyfunction)(void)` – wskaźnik do funkcji zawierającej kod zadania – w naszym przypadku są to funkcje o nazwach Task_1, Task_2 i Task_3,
- `u08 Pid` – 8–bitowa zmienna zawierająca numer zadania (zdefiniowane w *main.h*),
- `u08 u08Prio` – 8–bitowa zmienna zawierająca priorytet zadania – im mniejsza wartość, tym zadanie ma wyższy priorytet (zdefiniowane w *main.h*),
- `u16 u16StackSize` – zmienna o rozmiarze dwóch bajtów zawierająca rozmiar stosu zadania (zdefiniowane w *main.h*).

List. 2. Wywołanie funkcji tworzącej zadania w pliku *main.c*
```
KaRTOSTaskInit(&(Task_1),TASK_1_PID,TASK_1_PRIORITY,TASK_1_STACK) ;
KaRTOSTaskInit(&Task_2,TASK_2_PID,TASK_2_PRIORITY,TASK_2_STACK) ;
KaRTOSTaskInit(&Task_3,TASK_3_PID,TASK_3_PRIORITY,TASK_3_STACK) ;
//KaRTOSTaskInit(&Task_4,TASK_4_PID,TASK_4_PRIORITY,TASK_4_STACK)
```

Konfigurowanie systemu

W katalogu *Hello_world_tu_KaRTOS\KaRTOS\ATMega8* znajduje się plik *KaRTOS.conf*. W nim zawarte są zmienne konfigurujące system operacyjny. Plik jest podzielony na pięć sekcji, z których edytować będziemy pierwsze trzy:

1. Sekcja SWITCH ON/OFF KaRTOS MODULES – umożliwia włączenie lub wyłączenie poszczególnych modułów systemu do kompilacji. Dokonujemy tego „komentując" (lub nie) poszczególne zmienne kompilatora za pomocą znaków „//". Dla potrzeb naszej pierwszej aplikacji wszystkie moduły winny być wyłączone („zakomentowane") z wyjątkiem modułu obsługi portu szeregowego – KaRTOS_UART_ON.

2. Sekcja HARDWARE SYSTEM CONFIGURATION – w tej sekcji możemy ustawić takie parametry jak: rozmiar stosu systemowego (w bajtach) – SYS_STACK 20, podział pamięci RAM na sekcje: pamięci zmiennych globalnych oraz pamięci systemowej. NO_TASKS_RAM_ADDR 620, (więcej na ten temat w kolejnej części).

3. Sekcja KaRTOS_1MS_OCR_WART – wartość rejestru OCR timera systemowego zgłaszającego przerwanie co 1 ms. Częstotliwość zegara taktującego mikrokontroler jest dzielona wstępnie przez 64. W naszym wypadku ma wtedy wartość równą 8 [MHz] /64=125 [kHz]. Zatem aby odmierzyć 1/1000 sekundy, w rejestrze OCR musi się znajdować wartość 125. Dlatego: KaRTOS_1MS_OCR_WART=125.

Konfigurowanie pinów kontrolera

W katalogu *Hello_world_tu_KaRTOS\KaRTOS\ATMega8* otwieramy plik *Initm8.c*. Dokonujemy konfiguracji portów w zależności od tego, do których nóżek mikrokontrolera mamy podłączone diody. Jeśli nasza aplikacja ma działać w układzie z rys. 9, dokonujemy konfiguracji PORTU C następująco: `DDRC=0x03` oraz `PORTC=0x03` (piny 0 i 1 portu są wyjściami w stanie wysokim).

Implementacja kodu zadań

Na **list. 3** przedstawiono implementację zadania Task_1 realizującego wysyłanie danych przez port szeregowy. Jako pierwsza wywoływana jest funkcja systemowa *KaRTOSUar-*

List. 3. Implementacja zadania Task_1

```
void Task_1(void)
{
KaRTOSUartInit(12,0) ;          //-38400 bps dla 8, MHz
for(;;)
{
KaRTOSUartOpen() ;
KaRTOS_UART_PRINT(„\n\r*** Hello World tu KaRTOS !! ***”) ;
KaRTOSUartClose() ;
TimeSleepms(1000) ;
} ;
}
```

List. 4. Implementacja zadania Task_2

```
void Task_2(void)
{
for(;;)
{
cbi(PORTC,0) ;
TimeSleepms(1000) ;
sbi(PORTC,0) ;
TimeSleepms(1000) ;
} ;
}
```

List. 5. Implementacja zadania Task_3

```
void Task_3(void)
{
for(;;)
{
cbi(PORTC,1) ;
TimeSleepms(100) ;
sbi(PORTC,1) ;
TimeSleepms(400) ;
} ;
}
```

tInit dokonująca inicjalizacji portu USART mikrokontrolera. Deklaracja wspomnianej funkcji ma następującą postać: *void KaRTOSUartInit(u16 u16Baudrate, u08 u08doubleSpeed)*. Przyjmuje ona dwa parametry:

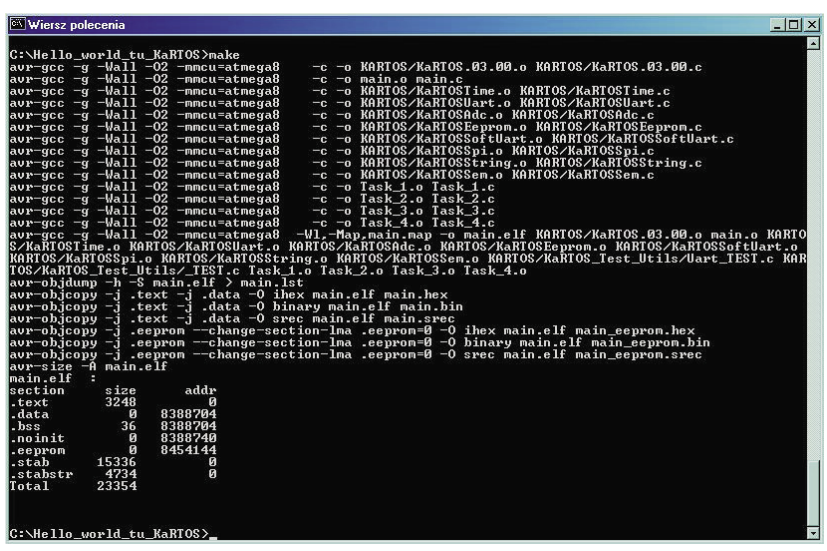
- `u16Baudrate` – wartość ta zostanie przepisana do rejestru UBRR, określa zatem prędkość transmisji,
- `u08doubleSpeed` – włącza (jeśli ma wartość 1) lub wyłącza (jeśli wartość jest różna od 1) podwojenie prędkości transmisji portu.

W naszym przypadku używamy zegara o częstotliwości 8 MHz, a chcemy uzyskać prędkość transmisji 38,4 kb/s bez podwojenia prędkości (`u08doubleSpeed=0`). Wykonując proste obliczenie podane w dokumentacji mikrokontrolera lub korzystając z gotowych tabelek również tam zawartych otrzymujemy wartość UBRR równą 12 (`u16Baudrate=12`).

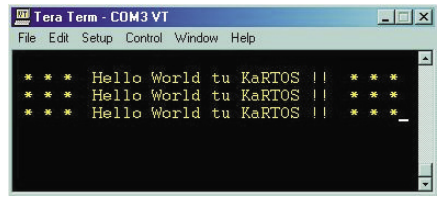
Kolejnym krokiem w naszym zadaniu (**list. 3**) jest zaimplementowanie nieskończonej pętli wysyłającej co sekundę ciąg znaków. Funkcja systemowa *KaRTOSUartOpen* otwiera port szeregowy, *KaRTOS_UART_PRINT* wysyła ciąg znaków będących jej argumentem i zawarty między znakami cudzysłów, a *KaRTOSUartClose* zamyka wcześniej otwarty port. Pozostaje nam jeszcze zrealizować sekundowe opóźnienie, do czego służy systemowa funkcja *TimeSleepms*. Jako argument podajemy czas w milisekundach – w naszym przypadku 1000 ms.

Zadania Task_2 i Task_3 będą miały podobną budowę ze względu na realizację podobnych algorytmów – różnią się jedynie czasami opóźnień. Kod zadań przedstawiają odpowiednio **list. 4** i **5**. Po ustawieniu odpowiedniego pinu w stan niski, co powoduje zapalenie podłączonej do niego diody, należy zrealizować opóźnienie o założonym czasie trwania. Ustawienie pinu w stan wysoki powoduje, że podłączona dioda gaśnie, a my znów realizujemy opóźnienie o odpowiednim czasie trwania. Realizując cyklicznie w nieskończonej pętli powyższy algorytm powodujemy mruganie diody z zadanymi czasami świecenia i nie świecenia.

Po dokonaniu wszystkich powyższych czynności jesteśmy gotowi do kompilacji kodu i zaprogramowania kontrolera.



Rys. 10. Widok okna konsoli po pomyślnej kompilacji aplikacji demo



Rys. 11. Widok okna terminala z działającą aplikacją

Kompilacja

Po rozpakowaniu archiwum *Hello_world_tu_KaRTOS.zip* na dysk *c:* możemy przystąpić do kompilacji. W tym celu otwieramy wiersz poleceń systemu windows i przechodzimy do katalogu *c:\Hello_world_tu_KaRTOS*. Po wydaniu polecenia make na konsoli powinny pojawić się komunikaty świadczące o postępie kompilacji i po chwili ekran powinien wyglądać jak na **rys. 10**. Oznacza to, że kompilacja przebiegła pomyślnie, a w katalogu *c:\Hello_world_tu_KaRTOS* znajduje się plik *main.hex* (i *main.bin*), z kodem wynikowym, który możemy załadować do pamięci kontrolera.

Aplikacja działa

Teraz wystarczy podłączyć sprzęt (rys. 9) do komputera PC za pomocą niekrosowanego kabla szeregowego i uruchomić dowolną aplikację terminala (np. TTermSSH). Następnie ustawiamy parametry transmisji, jak podano powyżej (8N1, 38400). Jeśli połączenia są wykonane poprawnie, to po podłączeniu zasilania do płyty kontrolera zaobserwujemy mrugające diody, a terminal zapełni się ekranem powitalnym jak na **rys. 11**.

Mariusz Żądło
iram@poczta.onet.pl

Autor zachęca Czytelników do kształtowania treści kolejnych odcinków cyklu. Napisz w e–mailu czy bardziej interesuje Cię teoria działania, czy raczej wolisz aby prezentowane były przykładowe aplikacje i projekty dla systemu KaRTOS. Inne uwagi również mile widziane.