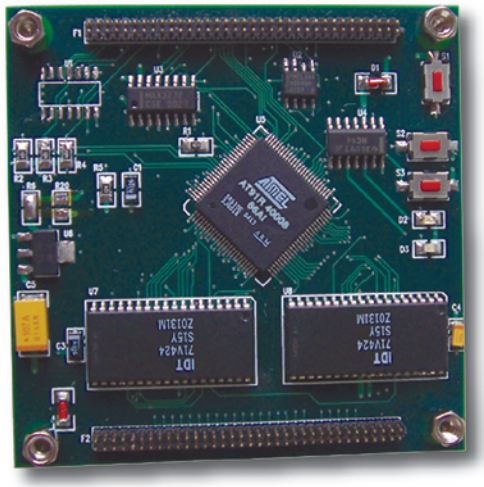
Wprowadzenie do systemów operacyjnych dla systemów wbudowanych na przykładzie platformy ARM, część 1

Popularne mikrokontrolery 32–bitowe, jak np. oparte na rdzeniu ARM7, używane w aplikacjach „wbudowanych” (embedded systems), mają stosunkowo dużą moc przetwarzania. Niestety, z różnych powodów zdarza się, że są one traktowane i wykorzystywane jak szybkie 8–bitowce, co (zwykle) jest marnowaniem ich zasobów. Aby w pełni wykorzystać możliwości przetwarzania nowoczesnego 32–bitowego mikrokontrolera (µC), warto pójść drogą, jaką rozwijały się „duże” maszyny, np. PC, a mianowicie użyć systemu operacyjnego (OS – operating system) do zarządzania zadaniami wykonywanymi przez µC, jak i jego zasobami. Warto przy tym pamiętać, że z powodu wymagań narzuconych przez specyficzne zastosowania, OS używany w systemach wbudowanych musi mieć specjalne właściwości w porównaniu np. z OS dla komputera biurkowego.

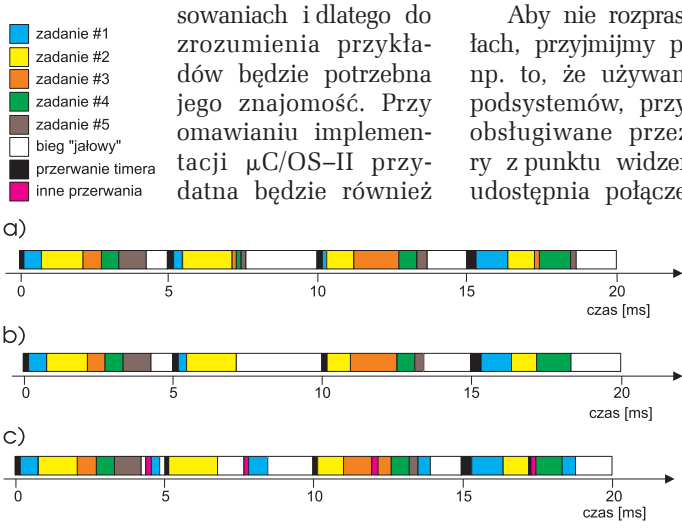


W pierwszej cyklu przedstawimy ewolucję struktury programu od prostej nieskończonej pętli do systemu wielozadaniowego z wywłaszczaniem. W celu uczynienia tego bardziej przystępnym rozważania będą bazować na przykładowej aplikacji – ministacji pogodowej. W dalszej części zostanie przedstawiony bardziej praktyczny aspekt zagadnienia, a mianowicie szczegóły implementacji wielozadaniowości z wywłaszczaniem w systemie czasu rzeczywistego (RTOS – *Real Time Operating System*) µC/OS–II na platformie mikrokontrolera AT91R40008 (rdzeń ARM7TDMI). Większość cytowanego kodu została napisana w języku C, który *de facto* stanowi standard w tego rodzaju zastosowaniach i dlatego do zrozumienia przykładów będzie potrzebna jego znajomość. Przy omawianiu implementacji µC/OS–II przydatna będzie również znajomość architektury i asemblera procesora ARM.

Przykładowa aplikacja ministacji pogodowej to oprogramowanie o następującej funkcjonalności:

- zbieranie danych z czujników zdalnych (przez łącze radiowe);
- zbieranie danych z czujników wewnętrznych (magistrala SPI);
- wyświetlanie informacji na wyświetlaczu LCD (sterowanym przez zwykłe porty I/O);
- sterowanie urządzeniem za pomocą klawiatury (np. zmiana trybu wyświetlania);
- udostępnianie danych z poziomu przeglądarki WWW – czyli prosty serwer WWW (protokoły Ethernet, TCP/IP, HTTP).

Aby nie rozpraszać się na szczegółach, przyjmijmy pewne uproszczenia, np. to, że używamy „inteligentnych” podsystemów, przykładowo łącze RF obsługiwane przez transceiver, który z punktu widzenia mikrokontrolera udostępnia połączenie radiowe przez zwykłe asynchroniczne łącze szeregowe (UART), dostęp do sieci zrealizowany jest za pomocą modułu mającego wbudowane złącze Ethernet i obsługujące protokół TCP/IP.



Rys. 1. Różne metody obsługi zadań: a) obsługa typu cycle scheduling, b) zmiana flag aktywacji przez zadania, c) zmiana flag aktywacji przez przerwania

Najprostszy sposób zaprojektowania struktury aplikacji dla takiego systemu to nieskończona pętla wykonywana cyklicznie. Podczas każdego przebiegu (cyklu) wykonuje się zadania odpowiedzialne za realizację poszczególnych funkcji urządzenia – jest to tzw. *cyclic scheduling* (**rys. 1a**, **list. 1**). Każde zadanie (*task*) to po prostu wywołanie funkcji języka C (uwaga: w dalszej części artykułu termin „zadanie” będzie używany zamiennie z „funkcją”, która realizuje funkcjonalność zadania).

Przy takim podejściu od razu napotykamy na pewne problemy związane jednak z naturą zadań. Część zadań jest typowo cykliczna (np. obsluz_wyswietlacz_lcd), część – zdarzeniowa (np. obsluz_serwer_http), inne natomiast łączą w sobie obie te cechy. Oczywiście, wszystkie zadania powinny być wykonywane dostatecznie szybko, tak aby nie blokować tych zadań, które akurat mają coś

List. 1. Aplikacja typu *times cycle scheduling*

```
int main(void)
{
  inicjalizacja();

  for ( ;; )
  {
    obsluga_klawiatury();
    odczyt_zdalny_czujnikow();
    odczyt_lokalny_czujnikow();
    obsluz_wyswietlacz_lcd();
    obsluz_serwer_http();

    czekaj_do_poczatku_nowego_cyk-
lu();
  }

  return 0;
}
```

List. 2. Obsługa zadań z podziałem na części

```
typedef Http_State_Tag
{
  idle,
  request,
  response
} Http_State_T;
static void obsluz_http(void)
{
  static Http_State_T state = idle;
  switch (state)
  {
  idle:
    ....
    state = request;
    break;
  request:
    ....
    state = response;
    break;
  response:
    ....
    state = idle;
    break;
    default:
    state = idle;
    break;
  }
}
```

„pilnego" do zrobienia, a dodatkowo cały czas trwania pętli głównej powinien być krótszy niż założony czas cyklu (w celu zapewnienia prawidłowego działania zadań cyklicznych/czasowych).

Wynika z tego, że zadania, które ze swojej natury trwają dłużej, należy (sztucznie) podzielić na części (podzadania) i w jednym przebiegu pętli głównej powinna być wykonywana tylko część zadania – podzadanie. Umożliwia to innym zadaniom odpowiednio częste „dochodzenie do głosu" i wykonywanie własnej części kodu (**list. 2**).

Pomysł nieskończonej cyklicznej pętli można dalej rozwijać na wiele różnych sposobów. Jednym z nich jest dodanie sprawdzania warunku (flagi aktywacji zadania) określającego, czy dane zadanie ma być wykonywanie w bieżącym obiegu pętli głównej. Flagi aktywacji mogą być zmieniane przez zadania (**rys. 1b**), jak i przerwania (**rys. 1c**), co daje pewną swobodę i dynamikę w sterowaniu wykonywaniem pętli głównej.

List. 3. Obsługa zadań z zagnieżdżonymi przerwaniami

```
void SPI_ISR(void)
{
  uint8_t rx_byte;

  rx_byte = spi_data_port();

  // odblokuj przerwania
  EI();

  // w tym momencie ta funkcja może
być przerwana przez obsługę innych
przerwań
  // UWAGA - dotyczy to także
obsługi kolejnego przerwania z ma-
gistrali SPI

  // tutaj następuje dalsza obsługa
bajtu odebranego z portu SPI
  ....

  // zakończenie ISR i powrót z prz-
erwania
}
```

W ten sposób doszliśmy do (uproszczonej) formy wielozadaniowości zwanej *cooperative multitasking*. Przełączanie pomiędzy wykonywanymi zadaniami odbywa się przez dobrowolne oddanie sterowania do „następnego" zadania. Konstrukcja opiera się na zasadzie zaufania, że żadna funkcja nie będzie działać „za długo". W przypadku niektórych urządzeń takie zaufanie to zdecydowanie za mało (osoby, które używały MS Windows 3.x, zapewne pamiętają, do czego to może prowadzić) i potrzeba sporo dodatkowej pracy, aby upewnić się, że system działa stabilnie w każdych warunkach.

Innym możliwym sposobem na poradzenie sobie z problemem zbyt długich zadań jest umieszczenie części funkcjonalności w procedurach obsługi przerwań (ISR – *Interrupt Service Routine*). Podejście to koliduje jednak z fundamentalną zasadą jak najkrótszego blokowania przerwań, czyli jak najszybszej ich obsługi. Oczywiście i ten problem można obejść przez umożliwienie zagnieżdżania przerwań. Na przykład, w kodzie ISR odpowiedzialnej za odbiór danych SPI, po odczytaniu rejestru danych z portu SPI wykonujemy instrukcję odblokowującą przerwania procesora i kontynuujemy przetwarzanie odebranych danych (np. umieszczamy je w buforze itp.) (**list. 3**).

Jednak takie pozornie proste rozwiązanie może nas szybko wpędzić w duże kłopoty. Po pierwsze, łatwo zapełnić stos (każde przerwanie zapisuje tam część rejestrów procesora i rezerwuje miejsce na zmienne lokalne ISR). Po drugie, nie eliminuje to do końca ograniczeń czasowych (których przecież chcemy uniknąć), ponieważ przenosimy problem tylko o jeden stopień wyżej, tj. do procedury obsługi przerwania. Na przykład, nadal trzeba zdążyć z obsługą przerwania, zanim wystąpi następne takie samo. Sprawy się dodatkowo komplikują w związku z tym, że przy takim rozwiązaniu nie da się przewidzieć, jak długo będzie trwała obsługa danego przerwania, ponieważ może być ona czasowo zawieszona przez obsługę przerwań zagnieżdżonych. Oczywiście, przy starannym i systematycznym podejściu jesteśmy w stanie przewidzieć i osza-

cować możliwe opóźnienia i czasy wykonania, ale jest to bardzo czasochłonne i tak naprawdę możliwe tylko przy mniejszych aplikacjach (można też użyć bardzo szybkiego procesora i liczyć na to, że zapas mocy obliczeniowej pozwoli na spełnienie wszystkich wymagań czasowych).

Powyższe rozwiązania są (w niektórych zastosowaniach) dobre, sprawdzone i często używane, ale ze swej natury bardzo statyczne i mało elastyczne. Znaczy to tyle, że po każdej, nawet drobnej zmianie funkcjonalności aplikacji powinniśmy pracowicie pomierzyć czasy wykonania (pod)zadań i przerwań, a następnie tak to wszystko ułożyć, aby spełnione były wymagania i zależności czasowe. Warto też zauważyć, iż rozwiązania takie zwykle są dalekie od optymalności pod względem użycia zasobów procesora, a to dlatego, że zawsze należy uwzględniać najdłuższe możliwe czasy wykonywania danego (pod)zadania (tzw. najgorszy możliwy przypadek – *worst case*), gdyż niestety „wolny" czas μC uzyskany dzięki temu, że któraś część kodu wykona się szybciej, nie może być zwykle wykorzystany do wykonania innego zadania.

Zamiast żmudnych analiz i uwzględniania wszystkich możliwych powiązań i zależności można zastosować tradycyjny informatyczny sposób radzenia sobie z problemami, a mianowicie wprowadzić dodatkowy poziom abstrakcji, czyli dodatkową „warstwę" (pośrednią), która ukryje część funkcjonalności i pozwoli skupić się na tym, co jest istotne dla danej aplikacji. W tym przypadku taką „warstwą" jest system operacyjny.

Warto zatem zastanowić się, jakie funkcje (usługi) powinna oferować „warstwa" OS–a, tak aby projektowanie i kodowanie aplikacji było ułatwione.

W naszym przykładowym urządzeniu ma działać pięć zadań:

1. Obsługa klawiatury – powinna być uruchamiana co 20 ms, tzw. *polling*; dopuszczalne jest opóźnienie w uruchomieniu o 5 ms.
2. Odczyt zdalnych czujników przez łącze RF – powinien być uruchamiany zdarzeniem polegającym na zapełnieniu bufora odbiorczego *transceivera*, czyli maksymalne opóźnienie to czas odbioru bajtu (tak aby nie stracić danych); załóżmy maksymalne opóźnienie 2 ms.
3. Odczyt czujników lokalnych przez SPI – powinien być uruchamiany co 30 s; opóźnienie rzędu 1 s jest w pełni akceptowalne.
4. Obsługa serwera HTTP – powinna być uruchamiana zdarzeniem polegającym na odebraniu/wysłaniu kolejnego pakietu bajtów lub po upływie określonego czasu (*timeout*); opóźnienie 200 ms jest dopuszczalne.
5. Obsługa wyświetlacza LCD – powinna być uruchamiana co 50 ms; opóźnienie 10 ms jest dopuszczalne.

Przykładowe koncepcje programu przedstawimy w drugiej części artykułu.

Artur Lipowski
Cezary Worek