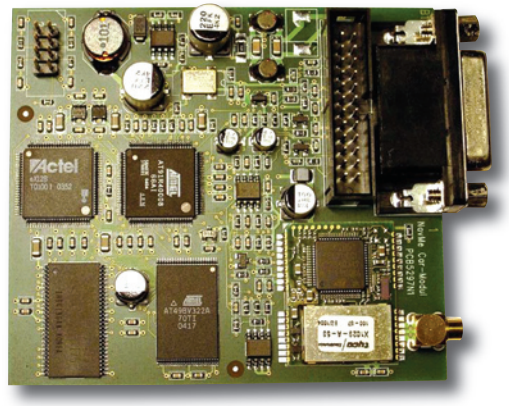
Wprowadzenie do systemów operacyjnych dla systemów wbudowanych na przykładzie platformy ARM, część 2

Kontynuujemy prezentację zagadnień związanych z działaniem systemów operacyjnych, ze szczególnym uwzględnieniem tych, które są stosowane w systemach embedded. Ten cykl jest dobrym przygotowaniem dla programistów zamierzających tworzyć oprogramowanie dla platform tego typu.



Poprzednią część artykułu zakończyliśmy opisem przykładowego projektu, który będzie implementowany na naszej hipotetycznej platformie. W tej części przedstawimy możliwe sposoby obsługi zadań tworzących nasz projekt. Podejdziemy do tematu „od góry" i zaprojektujemy program w sposób dla nas wygodny, zaczynając od kluczowych spraw i bez wdawania się w szczegóły.

Podejście 1

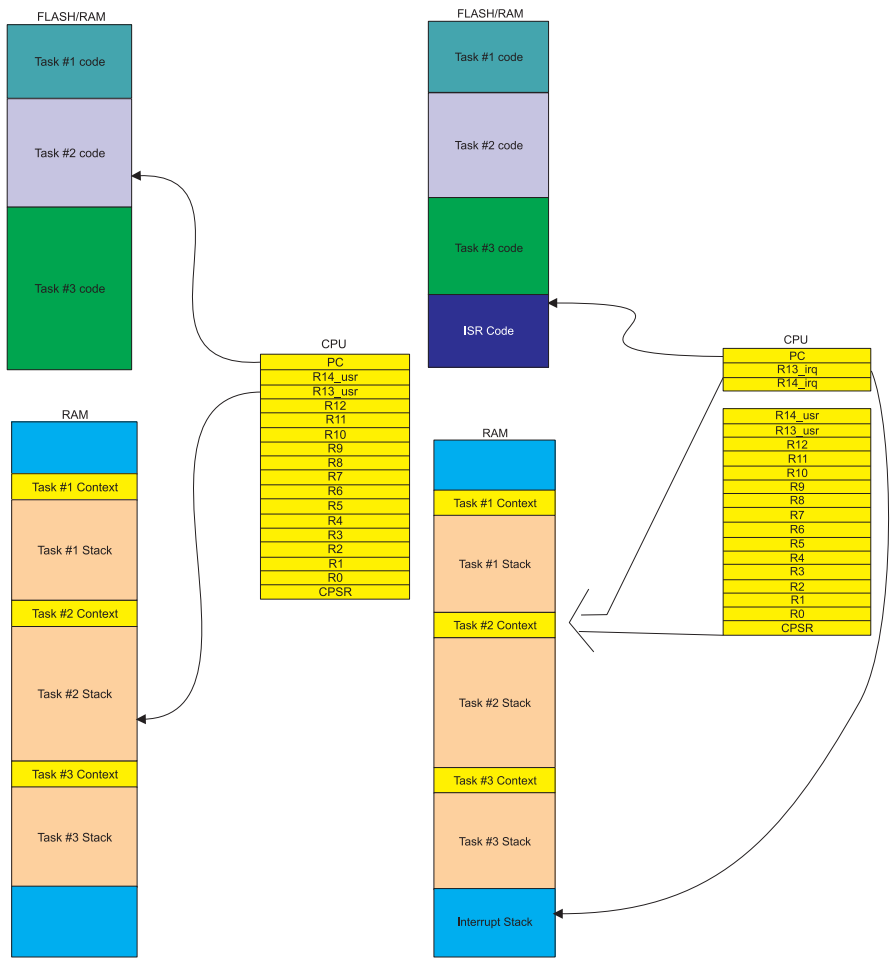
Mamy pięć mikrokontrolerów i na każdym działa jedno z powyższych zadań. Zadania mają możliwość komunikowania się ze sobą i wymieniania danych. Prawda, że szybko udało się zaprojektować cały system?

Podejście jest proste i skuteczne, jednak oprócz tych zalet ma wiele oczywistych wad. Spróbujmy zatem użyć techniki ostatnio bardzo modnej, czyli zastąpić funkcje sprzętowe (w tym przypadku cały mikrokontroler) przez oprogramowanie.

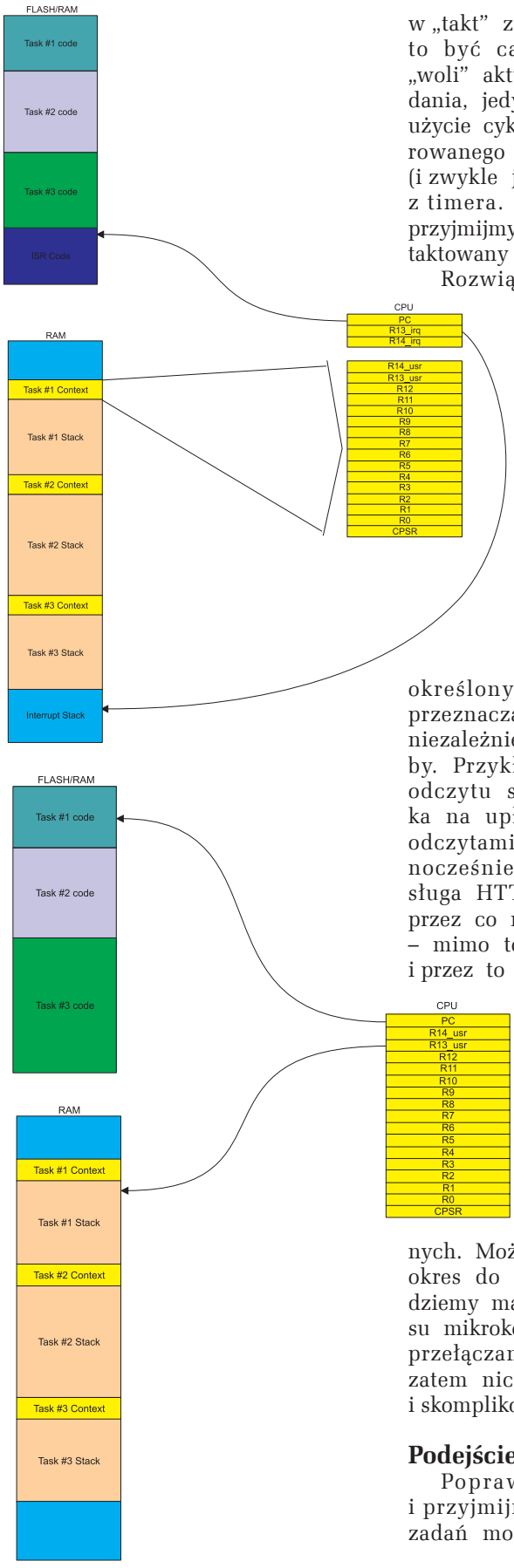
Podejście 2

Załóżmy, że mamy oprogramowanie zapewniające wymaganą funkcjonalność, czyli będące w stanie utworzyć środowisko, które daje wykonywanym zadaniom iluzję, że posiadają procesor na własność (działają na osobnych μC). W tym celu wystarczy (bagatela!), aby mikrokontroler wykonywał przez pewien krótki czas kod każdej z funkcji (zadania). Powiedzmy, najpierw przez 10 ms zadanie 1, potem przez 10 ms zadanie 2 itd., a po przejściu w ten sposób przez całą listę zadań wracamy do zadnia 1. Dodatkowo każde takie wykonanie kodu powinno odbywać się z załadowanym zestawem rejestrów procesora (w tym licznika rozkazów i wskaźnika stosu), „prywatnym" dla danej funkcji. Tego typu zapamiętany zestaw rejestrów jest częścią struktury danych, którą nazywamy kontekstem zadania. Jeżeli teraz kod każdego z zadań będzie uruchamiany odpowiednio często, to powstanie wrażenie, że wszystkie zadania są wykonywane jednocześnie (choć oczywiście wolniej, niż gdyby działały na osobnych mikrokontrolerach). Natomiast ze względu na to, że każde zadanie ma własny kontekst (w tym stos), który jest pamiętany w czasie, kiedy kod danego zadania nie jest wykonywany, zmienne lokalne (lokowane na stosie lub w rejestrach) są w każdej z funkcji (zadaniu) w pełni „prywatne" (**rys. 2a...2d**).

Przełączanie wykonywania kodu poszczególnych zadań odbywa się



Rys. 2a, b. Przykładowe realizacje obsługi wielu zadań - podejście 2



Rys. 2c, d. Przykładowe realizacje obsługi wielu zadań - podejście 2

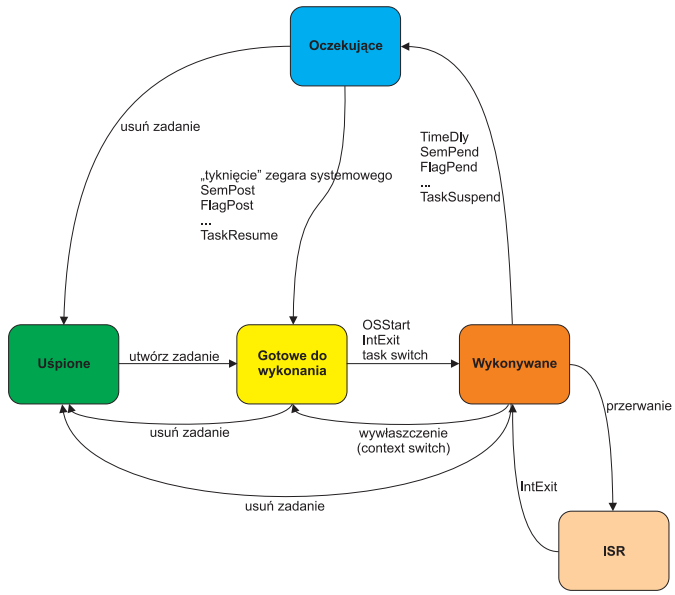
w „takt” zegara systemowego. Musi to być całkowicie niezależne od „woli” aktualnie wykonywanego zadania, jedynym więc sposobem jest użycie cyklicznego przerwania generowanego sprzętowo. Może to być (i zwykle jest) okresowe przerwanie z timera. Dla naszego przykładu przyjmijmy, że zegar systemowy jest taktowany z częstotliwością 100 Hz.

Rozwiązanie jest ciekawe i nie tak bardzo skomplikowane, ale właśnie ta prostota powoduje dużą nieefektywność. Z szybkiej analizy zadań i funkcjonalności naszego (i nie tylko) urządzenia wynika, że stanem normalnym jest to, iż w danej chwili powinna być wykonywana tylko część zadań (a czasem nawet żadne z nich), natomiast powyższe rozwiązanie implikuje to, że pewien określony czas procesora będzie przeznaczany dla każdego z zadań niezależnie od rzeczywistej potrzeby. Przykładowo, zadanie obsługi odczytu sensorów lokalnych czeka na upłynięcie czasu pomiędzy odczytami w „pustej” pętli, a jednocześnie inne zadanie (np. obsługa HTTP) potrzebuje procesora przez co najmniej 1 s bez przerwy – mimo to będzie ono przerywane i przez to opóźniane na rzecz zadania, które tak naprawdę nie ma nic „do roboty”, bo tylko oczekuje na upłynięcie określonego czasu. Co więcej, zegar systemowy ustawiony na 10 ms nie zapewnia spełnienia warunku czasu reakcji dla zadnia odczytu czujników zdalnych. Można oczywiście zmniejszyć okres do np. 1 ms, ale wtedy będziemy marnować bardzo dużo czasu mikrokontrolera tylko na funkcję przełączania zadań. Nie pozostaje zatem nic innego, jak rozbudować i skomplikować nasz OS.

Podejście 3

Poprawmy więc rozwiązanie i przyjmijmy, że każde z naszych zadań może być w jednym z czterech stanów (**rys. 3**):

1. Uśpione (*dormant*) – w tym stanie jest za-



Rys. 3. Realizacja obsługi wielu zadań - podejście 3

danie, które się już zakończyło lub jeszcze nie zostało przeznaczone do wykonywania; początkowo wszystkie zadania są w tym stanie.

2. Gotowe do wykonania (*ready*) – stan tuż po zażądaniu uruchomienia zadania lub po zakończeniu stanu oczekiwania.
3. Oczekujące (*waiting*) – stan, w którym funkcja (zadanie) czeka na spełnienie jakiegoś warunku, np. wystąpienie przerwania, upłynięcie określonego czasu, zwolnienie zasobu itp.
4. Wykonywane (*running*) – jedyne (w systemie jednoprocesorowym) zadanie, którego kod jest w danej chwili wykonywany.

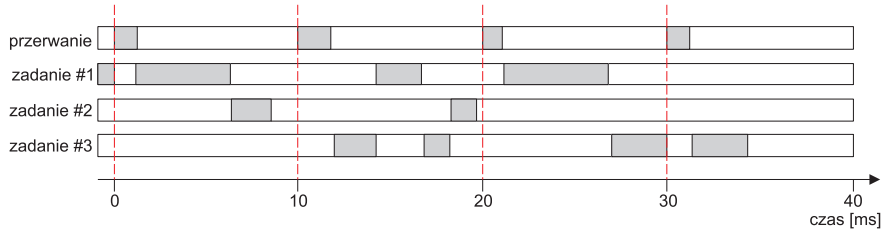
Co więcej, zakładamy też, iż przejścia pomiędzy powyższymi stanami, czyli faktyczne uruchomienie kodu zadania na procesorze, może odbywać się tylko według pewnych ściśle określonych reguł.

Jak zatem widać, całość zaczyna działać zdecydowanie bardziej efektywnie. Zadaniom oczekującym nie jest już przydzielany procesor. Dopiero po spełnieniu warunku zakończenia oczekiwania dane zadanie jest brane pod uwagę przy przydziale czasu procesora. Przełączanie kontekstu odbywa się też częściej, ponieważ teraz dokonuje się ono już nie tylko w przerwaniu zegara systemowego, lecz również wtedy, gdy zadanie przechodzi do stanu oczekiwania – wówczas automatycznie przestaje być ono wykonywane i następuje przełączenie do innego zdania (w stanie „gotowy do wykonania" **rys. 4**). Dla ułatwienia zadaniom szybszej reakcji na zdarzenia można też dokonywać jeszcze częstszego przełączania kontekstu zadań. Mogą być do tego celu wykorzystane inne przerwania oprócz zegara systemowego.

Pozostaje tylko jeden problem – co zrobić w przypadku, kiedy jest kilka zadań gotowych do wykonania? Które z nich wybrać?

Podejście 4

Funkcjonalność aplikacji wymaga, żeby niektóre zadania były wykonywane z większym reżimem czasowym (np. odbiór danych przez łącze RF) niż inne (np. lokalny odczyt



Rys. 4. Realizacja obsługi wielu zadań - podejście 4

sensorów). Dlatego dobrze byłoby uzależnić przydział czasu procesora od wagi (priorytetu) konkretnego zadania. W chwili, kiedy w stanie „gotowy do wykonania" jest więcej niż jedno zadanie, należałoby wybrać to spośród nich, które ma największy priorytet, i dokonać jego przełączenia do stanu wykonywania.

Dobrze jest w tym momencie wprowadzić ideę zadania specjalnego – *idle task*. Ma ono najmniejszy możliwy priorytet, istnieje cały czas (nie musimy go jawnie kodować i uruchamiać), nigdy nie przechodzi do stanu *waiting* i tak naprawdę działa wtedy, kiedy aplikacja „nic" nie robi (*idle*). Wbrew pozorom jest to niezwykle użyteczny pomysł. Ponieważ idle task ma najmniejszy priorytet spośród wszystkich zdań, będzie wykonywane wtedy i tylko wtedy, gdy **wszystkie** inne zadania nie mają nic do roboty (są w stanie *waiting*), a to z kolei pozwala na użycie *idle task* do (co najmniej) dwóch celów: np. pomiaru zużycia czasu mikrokontrolera, a w zależności od tego zarządzania poborem mocy (przez spowolnienie zegara lub przełączenie mikrokontrolera w tryb uśpienia).

Artur Lipowski
Cezary Worek