TinyOS 2.0

Nowa era programowania mikrourządzeń

Przemysław Horban i Jacek Migdał

Uniwersytet Warszawski

Zegarki eZ430-Chronos

- architektura MSP430
- pracujemy nad przeniesieniem TinyOS na tą platformę
- są one już zakupione i mamy ich dużo
- nasza praca pozwoli następnym studentom w pełni wykorzystać ich możliwości

Co potrafi taki zegarek i jaki jest stan prac

Demo

Hardware zegarka

- Procesor 20 MHz
- 4KB RAM
- 32KB flash (pamięć programu)
- Radio 868 MHz (transmisja 11 KB/s)
- Wyświetlacz LCD (4 + 5 cyfr oraz ikonki)
- Port szeregowy - UART (printf do PC itp.)
- Akcelerometr 3D
- Miernik ciśnienia
- 5 przycisków

Zasadniczy problem

????problem zasadniczy - czas życia na bateriach i zużycie energii - to decyduje o wszystskich kompromisach projektowych!???????????

Tradycyjny sposób ich programowania

??????????jak robiło się to zawsze - funkcja main(, wiele plików c, być może biblioteki. Piszemy swobodnie po rejestrach. Spore zamieszanie. Brak dostępnych abstrakcji. Piszesz po rejestrach - wady powyższego podejścia: nieprzenośne, każdy kontroler musi miec osobny code base, utrudnia zrobienie uniwersalnych stosów siecowych, masa błędów wielokrotnego włączaenia, konflikty sprzętowe, błędy dostępu do pamięci, śmieciaste aplikacje etc.)???

Typowy kod

??????????

Rozwiązanie tych problemów

- TinyOS jest systemem powstałym z myślą o tych problemach
- Aby im zaradzić potrzeba było nowych narzędzi - a w szczególności nowego języka: *nesC*

TinyOS 1.0

Zalety:

- nadaje aplikacji strukturę
- daje za darmo zarządzanie zdaniami i tym samym sposób myślenia na którym można się oprzeć
- dzięki interfejsom architektury HAA zapewnia przenośność wielu aplikacji między platformami
- pozwala łączyć ze sobą protokoły i algorytmy wynalezione przez różne grupy (LPL, CTP, Tricle)
- pozostawia wielką elastyczność programiście (to wciąż C)

Wady:

- duże problemy z zarządzaniem pamięcią (crash gdy jej zabraknie)
- częste błędy przy współbieżności, ze względu na przeploty z przerwaniami
- błędy podwójnej lub zapomnianej inicjalizacji
- trudne do ogarnięcia zależności między komponentami
- oddzielna kompilacja utrudnia statyczną analizę

Wnioski:

- można by dużo skorzystać, gdyby poczynić pewnie silniejsze założenia o języku programowania i zarządzaniu zasobami
- potrzeba większej hermetyzacji modułów oraz łatwiejszego sposobu organizowania zależności między modułami
- przydatna była by statyczna analiza kodu aplikacji
- ogólnie, bardzo potrzebne było lepsze wsparcie narzędziowe

nesC nadzieją na rozwiązanie powyższych problemów.

TinyOS 2.0

Najważniejsze cechy:

- rozwiązuje problemy rozpoznane przy pracy z TinyOS 1.0
- pamięć alokowana całkowicie statycznie, w czasie kompilacji
- statyczna analiza przeplotów wykonania współbieżnego generowanie ostrzeżeń w czasie kompilacji
- automatyczna inicjalizacja wciąganych komponentów
- statyczna analiza inicjalizacji (@once)
- przejrzyste granice między komponentami, wyznaczone przez interfejsy
- możliwość dogłębnej analizy i optymalizacji kodu
- wykorzystanie dorobku w dziedzinie kompilatorów

Pojęcie hermetycznego komponentu

- MainC
  - zawołam wszystkie metody init() interfejsów Init
  - zawołam potem zawołam metody boot() interfejsów Boot
  - wywołam pętlę zadań
- PlatformLCDC
  - podłączę wewnętrzną inicjalizację do MainC
  - daję interfejs LCDDriver
- SerialActiveMessageC
  - wezmę PlatformSerialC, który zapewni mi interfejs UartStream
  - włączę cały kod pakietowej obsługi portu szeregowego
  - dam Ci interfejs AMSend z metodą send()

Hermetyczne komponenty

- platforma dostarcza kilkanaście komponentów do obsługi urządzeń
- biblioteki TOS dostarczają swoje komponenty z usługami wyższego poziomu
- aplikacje (będące komponentami) dostarczają komponenty implementujące logikę

Bierzemy więc odpowiednie pudełka, łączymy je w konfiguracji i uzyskujemy działający program.

Wyświetlacz PC -> LCD

To jest aplikacja!

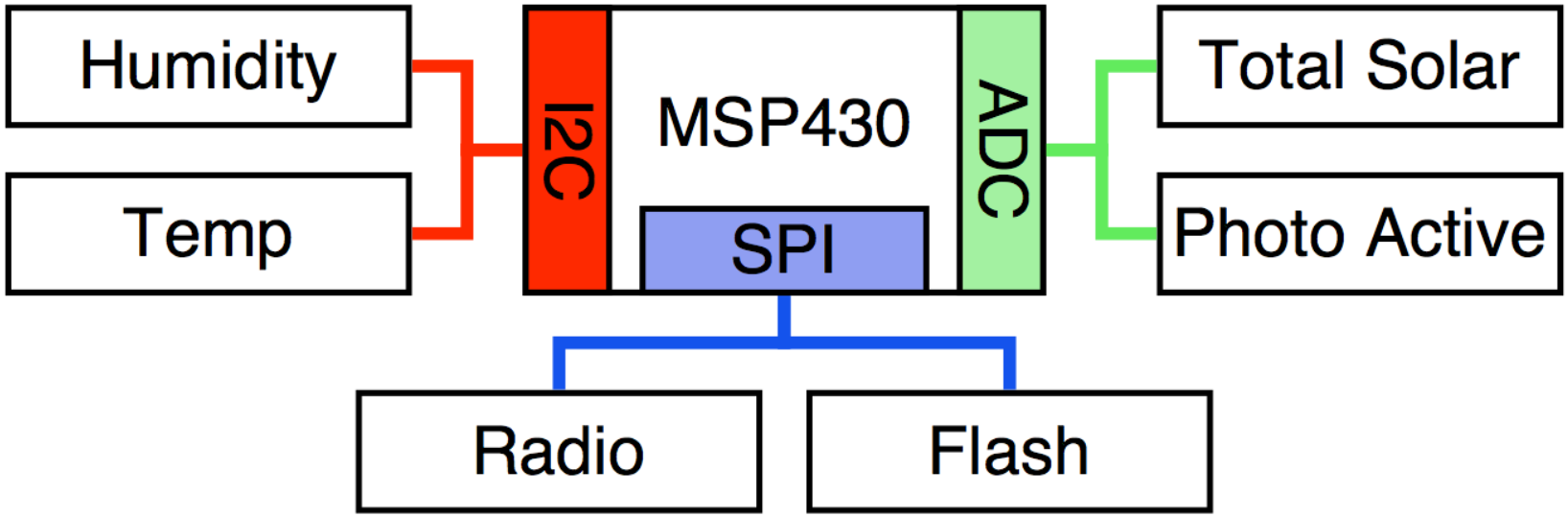
```
components MainC, SerialActiveMessageC;
components PlatformLCDC, PC2LCDAppP;
PC2LCDAppP.Boot -> MainC;
PC2LCDAppP.AMSend -> SerialActiveMessageC;
PC2LCDAppP.LCDDriver -> PlatformLCDC;
```

Case study: Virtualized timers

Pierwszy przykład - tyle zegarów ile tylko chcemy!

Case study: Integrated concurency and power management

Problem:



Case study: Integrated concurency and power management

- trzeba rozwiązać konflikty o zasoby
- ale również oszczędzać energię, gdy zasób nie jest używany

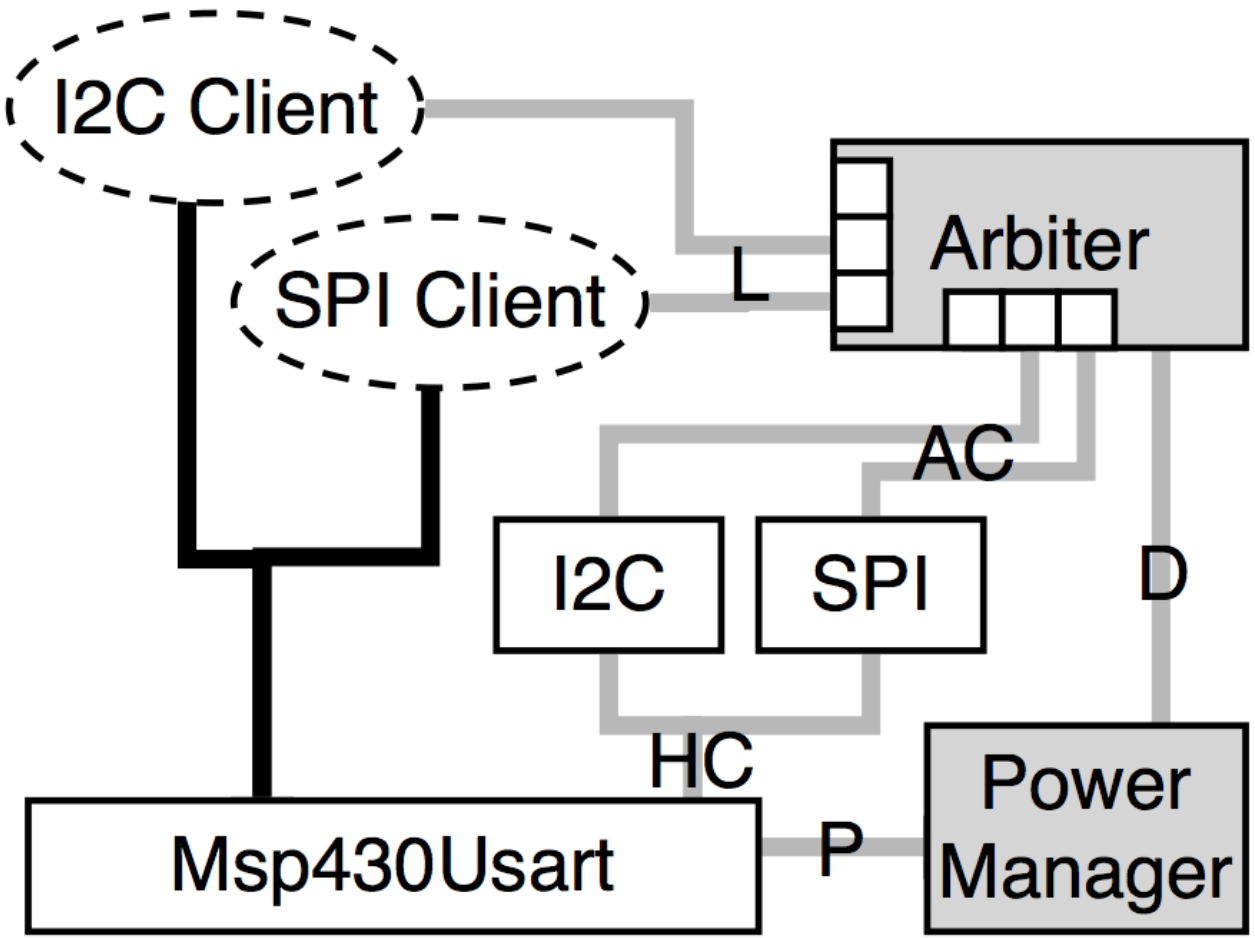
Okazuje się, że oba problemy są mają wspólne rozwiązanie.

Case study: Integrated concurency and power management

```
interface Resource {
   async command error_t request();
   event void granted();
   async command error_t release();
   async command bool isOwner();
}
```

Case study: Integrated concurency and power management

Rozwiązanie:



Case study: Zordon App

```
components MainC, ZordonP as App;
components PlatformLCDC;
components new AMSenderC(RANGER_CALL_MSG);
components new AMReceiverC(RANGER_CALL_MSG);
components ActiveMessageC;
App.Boot -> MainC;
App.LCDDriver -> PlatformLCDC;
App.AMSend -> AMSenderC;
App.Receive -> AMReceiverC;
App.AMControl -> ActiveMessageC;
```

Case study: Zordon App

```
components UpButtonC, DownButtonC;
components BacklightButtonC, StarButtonC;
App.NextNameButton -> UpButtonC;
App.PowerButton -> BacklightButtonC;
App.PrevNameButton -> DownButtonC;
App.SendButton -> StarButtonC;
components BeeperC;
App.Beeper -> BeeperC;
components TopLCDBlinkerC;
App.TopBlinkerControl -> TopLCDBlinkerC;
App.TopLCDBlinker -> TopLCDBlinkerC;
```