

TinyOS 2.0

Nowa era programowania mikroukładów

Przemysław Horban, Jacek Migdał

Uniwersytet Warszawski

Połączyliśmy w sieć komputery,
komórki... ale co z resztą
urządzeń?

Potrzebujemy nowej klasy urządzeń

Potrzebujemy nowej klasy urządzeń:

- tanich
- "gadających" po radiu
- wytrzymujących lata na bateriach.

Przykłady:

- Smart Grid
- sterowanie ruchem ulicznym
- monitorowanie zasobów

Przykłady:

- Smart Grid
- sterowanie ruchem ulicznym
- monitorowanie zasobów

Zegarek Chronoz ez-430:

- flash: 32 KB flash
- ram: 4 KB
- cpu: 20 MHz msp430
- sensory: akcelerometr, temperatura, ciśnienie
- radio: 867 MHz (90 metrów, 12 KB realnego transferu)
- dodatkowo: ekran LCD, przyciski

Jak programować

- Producent dołącza system chronos...
- W języku C z dodatkowymi definicjami.
- Wszystko trzeba robić ręcznie, ustawiać wartości itp.

- Pliki .c .h, które udostępniają parę funkcji
- Niestety są singletonem, nie mają żadnej kontroli dostępu.
- Problem braku inicjalizacji, podwójnej inicjalizacji, konflikt w dostępie do zasobów...

Zalety:

- nadaje aplikacji strukturę
- daje za darmo zarządzanie zdaniem 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, Trickle)
- 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 przerwaniem
- 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.

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

Część funkcjonalności TinyOS realizuje kompilator nesC:

- nesC: rozszerzenie C, jest też kompilowane do C
- wielowątkowość bez wywłaszczeń
- statyczna 'obiektość': na poziomie kompilacji

Organizacja kodu:

- dwukierunkowy zbiór sygnatur funkcji - interfejs
 - zdarzenie (event)
 - zadanie (task)
 - polecenie (command)
- kod C - moduły
 - wykorzystuję i dostarcza interfejsów
- łączenie wielu komponentów w jeden - konfiguracja

Pojęcie hermetycznego komponentu

- MainC
 - zwołam wszystkie metody init() interfejsów Init
 - zwołam potem zwołam metody boot() interfejsów Boot
 - wywołam pętlę zadań
- PlatformLCDC
 - podłączę wewnętrzną inicjalizację do MainC
 - dam 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!

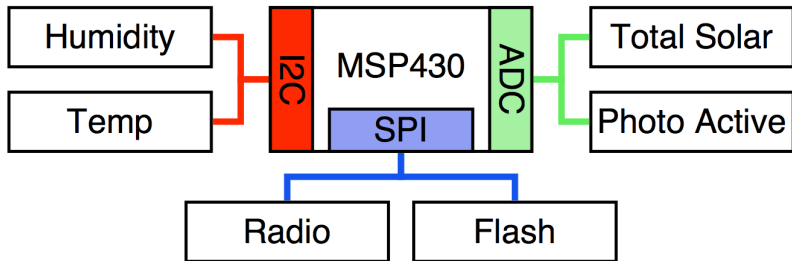
```
components MainC, SerialActiveMessageC;  
components PlatformLCDC, PC2LCDAppP;  
PC2LCDAppP.Boot -> MainC;  
PC2LCDAppP.AMSend -> SerialActiveMessageC;  
PC2LCDAppP.LCDDriver -> PlatformLCDC;
```

Case study: Virtualized timers

Pierwszy przykład - tyle zegarów ile tylko chcemy!

Case study: Integrated concurrency and power management

Problem:



Case study: Integrated concurrency and power management

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

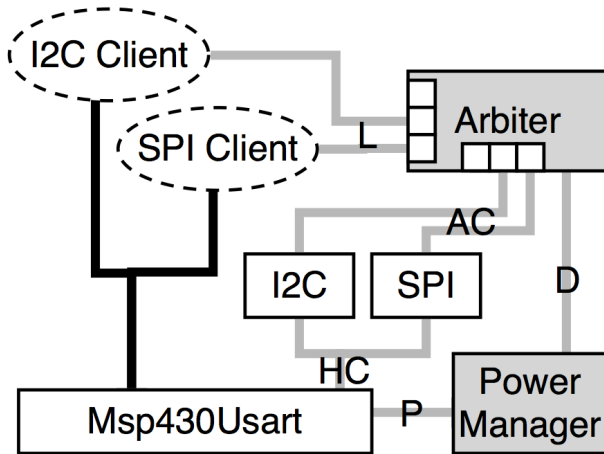
Okazuje się, że oba problemy są mają wspólne rozwiązanie.

Case study: Integrated concurrency 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 concurrency 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;
```