

TinyOS 2.0

Nowa era programowania mikrouządzeń

Przemysław Horban, 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 (sterowniki, wygodne środowisko programistyczne)

Demo

- 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

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

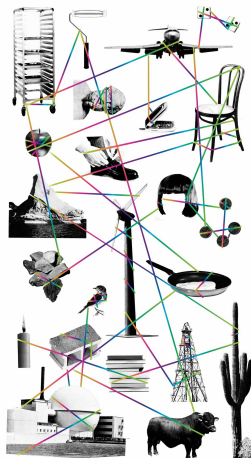
Potrzebujemy nowej klasy urządzeń

Potrzebujemy nowej klasy urządzeń:

- tanich
- "gadających" po radiu
- wytrzymujących lata na bateriach - ultra niski duty cycle

Przykłady:

- Smart Grid
- sterowanie ruchem ulicznym
- monitorowanie zasobów



Jak programować

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

Przykład kodu Texas Instruments

```
24 #include "msp430x22x4.h"
25
26 void main(void)
27 {
28     WDTCTL = WDT_ADLY_250;
29     IE1 |= WDTIE;
30     P1DIR |= 0x01;
31
32     __bis_SR_register(LPM3_bits + GIE);
33 }
34
35 // Watchdog Timer interrupt service routine
36 #pragma vector = WDT_VECTOR
37 __interrupt void watchdog_timer(void)
```

- 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...

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

Część funkcjonalności TinyOS realizuje kompilator nesC:

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

Organizacja kodu:

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

Jak to działa?

- Gdzieś w kodzie (komponent MainC) startuje funkcja `main()` - dla nas niewidoczna
- Wywołuje ona metody interfejsów `Init` i `Boot`
- W ten sposób inicjalizowane są urządzenia i dane aplikacji
- Dalej wykonują się już tylko przerwania i pętla schedulera
- Przerwania generuje np. zegar (*event Timer.fired()*)
- Task dla schedulera to funkcja *void f(void)*
- Kod, np. w trakcie obsługi przerwania, zleca wykonanie tasku wywołaniem *post zadanie()*
- Praca TinyOS to ciągła obsługa przerwań i wykonywanie task'ów

Najważniejsze zalety:

- Rozwiązuje problemy zarządzania pamięcią dzięki całkowicie statycznej alokacji
- Dbą o poprawną inicjalizację użytych komponentów (@once i automatyzacja)
- Porządkuje wykonanie głównego (scheduler) i obsługę przerw
- Zapewnia statyczną analizę przepływów i ostrzega o wyścigach do zasobów
- Dostarcza spójnych i hermetycznych komponentów, które są łatwe w użyciu
- Bardzo upraszcza zarządzanie zależnościami między komponentami
- Umożliwia statyczną analizę kodu całej aplikacji
- Nie marnuje dorobku w zakresie kompilatorów

Zalety związane ze strukturą kodu:

- Ułatwia myślenie i projektowanie dostarczając gotowej struktury systemu
- Dostarcza i wspiera wiele platform współdzielących jedną bazę kodu
- Umożliwia powstawanie interfejsów wysokiego poziomu (HAA)
- Sprawia, że implementacje algorytmów opracowywanych przez różne grupy mogą być użyte w jednej aplikacji
 - Low power listening
 - TRICLE
 - Collection tree protocol
- Dostarcza wielu bibliotek, łatwych i gotowych do użycia przy nowych platformach
- Ma bardzo przyjazną licencję

Pojęcie hermetycznego komponentu

Jest to coś co bardzo ułatwia pracę programisty.

- MainC
 - woła metody init() interfejsów Init
 - woła metody boot() interfejsów Boot
 - wchodzi pętlę task schedulera
- PlatformLCDC
 - podłącza wewnętrzną inicjalizację do MainC
 - inicjalizuje kontroler LCD
 - dostarcza interfejs LCDDriver
- SerialActiveMessageC
 - bierze z platformy PlatformSerialC, który dostarcza interfejs UartStream
 - podłącza kod pakietowej obsługi portu szeregowego
 - dostarcza m. in. interfejs AMSend z metodą send()

- Platforma dostarcza kilkanaście komponentów do obsługi urządzeń (np. strumień bajtów)
- Biblioteki TOS dostarczają swoje komponenty z usługami wyższego poziomu (np. transmisja pakietowa)
- Aplikacje dostarczają moduły implementujące logikę aplikacji

Elementy te zbierane są razem i łączone za pomocą konfiguracji, w efekcie czego uzyskujemy działający program.

Przykład: 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

Problem:

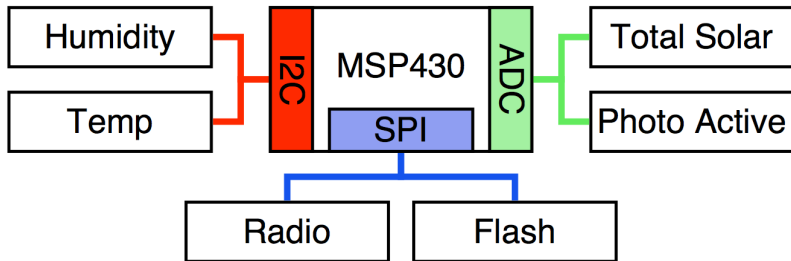
- Sterowanie urządzeniem wymaga zegarów. Potrzebujemy ich dużo.
- Mamy ich mało (zazwyczaj 3).

Rozwiązanie:

- Wirtualizacja:
 - TinyOS dostarcza generyczny komponent zegara: TimerMilliC
 - Możemy mieć wiele instancji jego i używać go jako zegara sprzętowego.
 - Jest Singleton, który kontroluje dostęp do sprzętu.
- Alokujemy dokładnie tyle zasobów ile potrzeba.
- Unikamy niepotrzebnych zależności pomiędzy komponentami.

Case study: Integrated concurrency and power management

Problem:



Case study: Integrated concurrency and power management

Problem:

- Kilka szyn komunikacyjnych
- Każda ma kilka urządzeń, ale obsługiwać może tylko jedno
- Wszystko trzeba jeszcze włączać i wyłączać

Case study: Integrated concurrency and power management

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

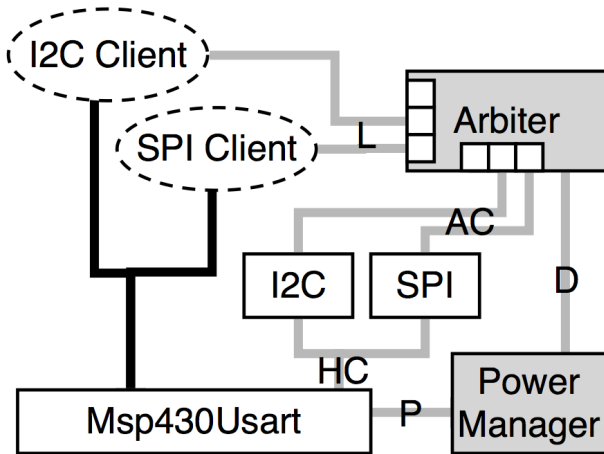
Okazuje się, że oba problemy są mają wspólne rozwiązanie.

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: Integrated concurrency and power management

- Klient prosi o zasób
- Arbiter rozpatruje zapytania
- Jeśli urządzenie wyłączone, konfiguruje je
- Power manager dba by procesor nie przeszedł w zbyt głęboki stan uśpienia
- Klient dostaje urządzenie gotowe do działania
- Potem je zwalnia
- Gdy zabraknie już rządań urządzenie jest wyłączane

Case study: Integrated concurrency and power management

- Osąga to efektywność bliską 99% ręcznie dostrojonym rozwiązaniom.
- Pamiętajmy, że chcemy możliwie skrócić czas gdy prądożerne urządzenia są włączone, więc ciągle coś włączamy i wyłączamy - a tu takiego kodu prawie nie ma

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;
```

- Ważnym problemem jest to, że aplikacje jest dość trudno debugować
- Mamy funkcję printf
- Dla Chronos'a działa nawet debugger w Eclipse!

Najczęstszy przypadek jest i tak taki: Miała świecić dioda.
Wgrywamy - nie świeci. I co?
Piotr opowie więcej jak temu zaradzić.