

LABORATORUL 03

Sisteme de operare în timp real: firmware pe un ceas open source

Durată: 4 ore | Suport: Capitolele 7 și 8 | Platformă: PineTime + InfiniTime (FreeRTOS) | Carte capitolul de referință

CUPRINS

1.	Obiectivele lucrării
2.	Ce construim
3.	Materiale necesare
4.	De ce nu ajunge o buclă principală
5.	Simulator de planificare
6.	Procese FreeRTOS în practică
7.	Comunicarea între procese
8.	Inversiunea de prioritate
9.	Compilarea firmware-ului InfiniTime
10.	Adăugarea unui proces propriu
11.	Încărcarea pe ceas
12.	Sarcini de lucru
13.	Provocare
14.	Întrebări de verificare
15.	Resurse

Până acum am scris programe care fac un singur lucru. Un ceas inteligent trebuie să deseneze ecranul, să citească accelerometrul, să întrețină conexiunea Bluetooth și să numere secunde - simultan, pe un singur procesor, cu 64 KB de memorie și o baterie care trebuie să țină o săptămână. În această lucrare studiem cum se rezolvă asta, pe un produs real al cărui cod sursă este public integral.

1 Obiectivele lucrării

- ☐ Înțelegerea diferenței dintre o buclă principală și un planificator preemptiv
- ☐ Aplicarea criteriilor de planificabilitate: utilizare, limita Rate Monotonic, EDF
- ☐ Compilarea unui firmware complet, real, din codul sursă
- ☐ Crearea unui proces FreeRTOS propriu și alegerea corectă a priorității și a stivei
- ☐ Comunicarea între procese prin cozi de mesaje, în locul variabilelor globale
- ☐ Provocarea și rezolvarea inversiunii de prioritate
- ☐ Legătura dintre planificare și consum: de ce un proces prost scris golește bateria

2 Ce construim

Luăm firmware-ul **InfiniTime** - sistemul care rulează pe ceasul PineTime - îl compilăm din sursă, îi adăugăm un proces propriu și un cadran nou, apoi îl încărcăm pe ceas prin Bluetooth. La final măsurăm ce a costat, în consum, ceea ce am adăugat.

DE CE TOCMAI ACEST CEAS

PineTime este singurul ceas inteligent la care aveți acces la **ABSOLUT TOT**

: schema electrică, cablajul, codul bootloaderului și codul firmware-ului. Nu este un exercițiu didactic - este un produs care se vinde, cu constrângeri reale de memorie și de baterie. Puteți citi de ce a fost luată fiecare decizie de proiectare, apoi o puteți schimba și vedea ce se întâmplă. Acest lucru nu este posibil pe niciun ceas comercial.

PROCESOR

nRF52832

ARM Cortex-M4F, 64 MHz

MEMORIE

64 KB RAM · 512 KB flash

plus 4 MB flash extern pentru resurse

ECRAN

ST7789, 240×240

IPS color, cu panou tactil capacitiv

SENZORI

accelerometru · puls

RADIO

Bluetooth Low Energy

BATERIE

180 mAh

BMA421 și HRS3300

și pentru actualizarea firmware-ului

autonomie de aproximativ o săptămână

CONSTRÂNGEREA CARE SCHIMBĂ TOTUL

64 KB de memorie înseamnă că un singur proces cu stiva prost dimensionată poate doborî întreg sistemul. Pe un calculator obișnuit ați alocat liniștit un tampon de 100 KB; aici nu aveți atâta memorie în total. Fiecare proces din InfiniTime are stiva măsurată la octet - și veți vedea cum.

3 Materiale necesare

1

PineTime (variantea sigilată e suficientă)

1

Calculator cu Linux sau WSL2

1

Docker instalat

1

Telefon Android cu Gadgetbridge (pentru încărcarea prin Bluetooth)

1

Cablu de încărcare magnetic (inclus cu ceasul)

1

ESP32-C6 din lucrarea precedentă (pentru partea de FreeRTOS fără ceas)

LUCRAREA FUNCȚIONEAZĂ ȘI FĂRĂ CEAS FIZIC

Secțiunile 5, 6 și 9 (planificare, procese, inversiune de prioritate) se pot parcurge integral pe ESP32-C6, care rulează tot FreeRTOS. Secțiunea 7 include și

INFINISIM

, un simulator care rulează firmware-ul complet pe calculator, cu ecran și atingere emulate. Ceasul fizic este necesar doar pentru secțiunile 10 și 11.

4 De ce nu ajunge o buclă principală

Un program de microcontroler începe aproape întotdeauna așa:

bucla clasică - și problema ei

```
void loop() {
  citeste_accelerometru();    // 2 ms
  actualizeaza_ecranul();      // 30 ms <-- lung
  trateaza_bluetooth();       // 5 ms
  numara_secunde();           // 1 ms
}
```

Funcționează până în clipa în care una dintre sarcini devine lentă. Când timp actualizeaza_ecranul() desenează, celelalte trei nu se execută. Dacă un cadru Bluetooth sosește în acest interval, se pierde. Iar dacă ecranul are nevoie ocazional de 200 ms, ceasul rămâne în urmă vizibil.

Se poate cârpi cu mașini de stare și verificări de millis() - exact tehnica de la laboratorul de microcontrolere. Dar la a cincea sau a șasea sarcină, codul devine imposibil de urmărit. Soluția industrială este un **planificator** care întrerupe o sarcină la mijloc și dă procesorul alteia, mai urgente.

Vocabularul planificării

Termen	Notăție	Înțeles
Perioadă	T	la ce interval trebuie reluată sarcina
Durată de execuție	C	cât timp de procesor îi trebuie, în cazul cel mai defavorabil
Termen limită	D	până când trebuie terminată; de obicei $D = T$
Utilizare	$U = \sum C/T$	ce fracțiune din procesor este cerută în total
Preempțiune	-	întreruperea unei sarcini în curs, în favoarea uneia mai urgente

Cele două politici clasice

	Rate Monotonic (RM)	Earliest Deadline First (EDF)
Prioritatea	fixă: perioadă mai mică → prioritate mai mare	dinamică: termenul cel mai apropiat câștigă
Garanție	$U \leq n \cdot (2^{1/n} - 1)$	$U \leq 1$
Limita pentru n mare	≈ 69 %	100 %
Cost la execuție	mic - prioritățile se calculează o dată	mai mare - termenele se recompară permanent
Comportare la supraîncărcare	previzibilă: procesele lente cedează primele	imprevizibilă: poate rata totul
Folosit de	FreeRTOS, Zephyr, majoritatea sistemelor încorporate	sisteme de cercetare, unele nuclee Linux în timp real

TESTUL DE UTILIZARE ESTE SUFICIENT, NU NECESAR

Dacă U este sub limita RM, sistemul este garantat planificabil. Dacă o depășește, **NU ÎNSEAMNĂ CĂ NU FUNCȚIONEAZĂ**

- înseamnă doar că acest test nu poate decide, iar răspunsul trebuie căutat prin simulare sau prin analiza timpului de răspuns. Este o distincție pe care simulatorul de mai jos o face explicită.

5 Simulator de planificare

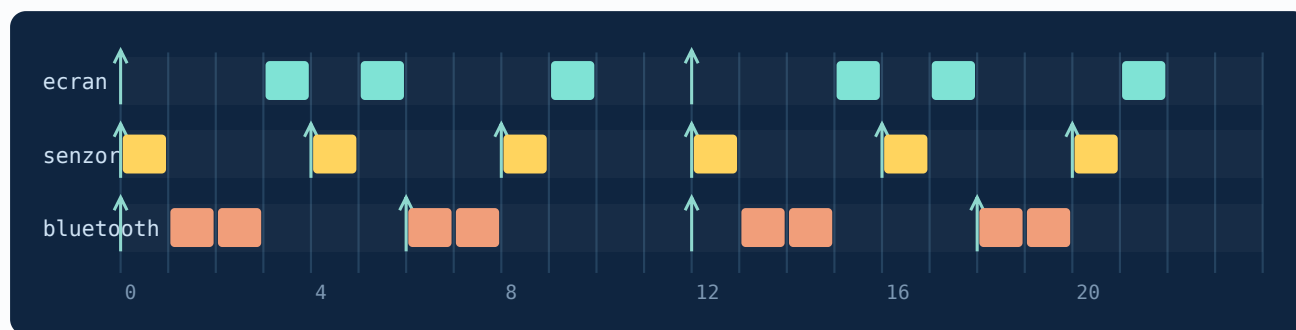
Trei procese periodice, un procesor. Mișcați cursoarele pentru durata de execuție și urmăriți diagrama: săgețile în sus sunt momentele de lansare, dreptunghiurile colorate sunt execuția efectivă, iar un **×** roșu marchează un termen ratat. Comutați între cele două politici și comparați.

► RATE MONOTONIC ȘI EDF, CU DIAGRAMĂ DE EXECUȚIE

ALGORITM

PROCES	DURATĂ C	PERIOADĂ T	UTILIZARE
■ ecran	3	12	25.0 %
■ senzor	1	4	25.0 %
■ bluetooth	2	6	33.3 %

■ ecran (C=3, T=12) ■ senzor (C=1, T=4) ■ bluetooth (C=2, T=6)



U = 83.3 % depășește limita RM de 78.0 %, deci testul de utilizare nu poate decide. Simularea nu arată însă nicio ratare: testul e suficient, nu necesar.

Versiune interactivă pe site.

TREI EXPERIMENTE DE FĂCUT, ÎN ORDINE

1. GĂSIȚI LIMITA RM.

Creșteți durata procesului „ecran” treaptă cu treaptă. La ce utilizare apare primul termen ratat? Comparați cu limita teoretică de 78 % pentru trei procese.

2. ZONA INCERTĂ.

Aduceți utilizarea între limita RM și 100 %. Verdictul devine „testul nu poate decide”, iar simularea arată totuși zero ratări. Aceasta este dovada practică a faptului că testul este suficient, dar nu necesar.

3. SUPERIORITATEA EDF.

Găsiți o configurație la care RM ratează, apoi comutați pe EDF fără să schimbați nimic altceva. Aceleași procese, același procesor, alt rezultat.

DE CE INDUSTRIA ALEGE TOTUȘI RM

EDF folosește procesorul până la ultimul procent, dar are două neajunsuri grave. Costă mai mult la execuție, pentru că termenele trebuie recomputeate la fiecare comutare. Și, mai important, la supraîncărcare se prăbușește haotic: nu se poate prezice ce proces va rata. Cu RM, la supraîncărcare cedează întotdeauna procesele cu perioada cea mai mare - deci proiectantul poate hotărî dinainte ce anume are voie să sufere.

6 Procese FreeRTOS în practică

Codul de mai jos rulează neschimbat pe ESP32-C6 și, cu modificări minime, pe orice sistem FreeRTOS - inclusiv pe PineTime.

procese_baza.ino - trei procese periodice

```
// Prioritatile FreeRTOS: numar mai MARE = prioritate mai MARE.
// Le alegem dupa regula Rate Monotonic: perioada mai mica, prioritate mai mare.
#define PRIO_SENZOR      3          // perioada 20 ms
#define PRIO_BLUETOOTH    2          // perioada 100 ms
#define PRIO_ECRAN       1          // perioada 200 ms

void procesSenzor(void* param) {
    // vTaskDelayUntil pastreaza o perioada EXACTA, indiferent cat a durat lucrul.
    TickType_t ultima = xTaskGetTickCount();
    for (;;) {
        citeste_accelerometru();
        vTaskDelayUntil(&ultima, pdMS_TO_TICKS(20));
    }
}

void procesBluetooth(void* param) {
    TickType_t ultima = xTaskGetTickCount();
    for (;;) {
        trateaza_bluetooth();
        vTaskDelayUntil(&ultima, pdMS_TO_TICKS(100));
    }
}

void procesEcran(void* param) {
    TickType_t ultima = xTaskGetTickCount();
    for (;;) {
        deseneaza_ecranul();          // poate dura 30 ms - va fi intrerupt de senzor
        vTaskDelayUntil(&ultima, pdMS_TO_TICKS(200));
    }
}

void setup() {
    Serial.begin(115200);

    //          functia      nume      stiva  param  prioritate  descriptor
    xTaskCreate(procesSenzor, "senzor", 2048, NULL, PRIO_SENZOR, NULL);
    xTaskCreate(procesBluetooth, "bluetooth", 4096, NULL, PRIO_BLUETOOTH, NULL);
    xTaskCreate(procesEcran, "ecran", 4096, NULL, PRIO_ECRAN, NULL);
}

void loop() {
    // Bucla principala este ea insasi un proces FreeRTOS, cu prioritate 1.
```

```
// 0 lasam sa doarma, ca sa nu fure timp de procesor.
vTaskDelay(pdMS_TO_TICKS(1000));
}
```

Trei greșeli care se fac aproape întotdeauna

1. VTASKDELAY ÎN LOC DE VTASKDELAYUNTIL

vTaskDelay(20) înseamnă „așteaptă 20 ms
DE ACUM

”. Dacă lucrul a durat 5 ms, perioada reală devine 25 ms, iar eroarea se acumulează. vTaskDelayUntil se raportează la momentul precedent de lansare și menține perioada exactă. Pentru orice sarcină periodică, a doua variantă este cea corectă.

2. DELAY() ÎNTR-UN PROCES

Pe ESP32, delay() este redirectat către vTaskDelay() și se comportă cuminte. Pe alte sisteme FreeRTOS este o buclă de așteptare activă care

BLOCHEAZĂ TOATE PROCESELE DE PRIORITATE MAI MICĂ

pe toată durata ei. Folosiți întotdeauna funcțiile FreeRTOS.

3. STIVA ALEASĂ LA ÎNTÂMPLARE

Fiecare proces primește propria stivă, iar depășirea ei nu dă o eroare clară: sistemul se blochează sau repornește aparent aleatoriu, adesea abia după câteva ore. Măsurați, nu ghiciți:

verificarea stivei

```
void procesDiagnostic(void* param) {
    for (;;) {
        // Intoarce numarul minim de cuvinte ramase LIBERE de la pornire.
        // Daca se apropie de zero, stiva este subdimensionata.
        UBaseType_t liber = uxTaskGetStackHighWaterMark(NULL);
        Serial.printf("stiva ramasa: %u cuvinte (%u octeti)\n",
                      liber, liber * sizeof(StackType_t));
        vTaskDelay(pdMS_TO_TICKS(5000));
    }
}
```

REGULA PRACTICĂ

Rulați aplicația cu toate ramurile de cod solicitate (inclusiv cazurile de eroare), citiți valoarea minimă rămasă, apoi păstrați o rezervă de aproximativ 30 %. Pe PineTime, unde memoria totală este de 64 KB, această măsurătoare nu este o rafinare - este singurul mod de a face aplicația să încapă.

7 Comunicarea între procese

Două procese care scriu în aceeași variabilă globală produc coruperi de date greu de reprodus. Soluția FreeRTOS este **coda de mesaje**: un tampon circular protejat, în care un proces depune și altul extrage, fără ca ele să se cunoască.

coada.ino - de la senzor la ecran

```

typedef struct {
    uint32_t moment_ms;
    float    x, y, z;
} Masuratoare;

QueueHandle_t coada;

void procesSenzor(void* param) {
    TickType_t ultima = xTaskGetTickCount();
    for (;;) {
        Masuratoare m = { millis(), citesteX(), citesteY(), citesteZ() };

        // Al treilea parametru este cat asteptam daca coada e plina.
        // Cu 0 nu asteptam deloc: mai bine pierdem o masuratoare decat sa
        // intarziem un proces de prioritate mare.
        if (xQueueSend(coada, &m, 0) != pdTRUE) {
            // Coada plina inseamna ca cititorul nu tine pasul - o informatie utila.
            Serial.println("masuratoare pierduta - consumatorul e prea lent");
        }
        vTaskDelayUntil(&ultima, pdMS_TO_TICKS(20));
    }
}

void procesEcran(void* param) {
    Masuratoare m;
    for (;;) {
        // portMAX_DELAY: procesul doarme pana soseste ceva. Cat doarme,
        // NU consuma timp de procesor - exact ce ne trebuie pentru baterie.
        if (xQueueReceive(coada, &m, portMAX_DELAY) == pdTRUE) {
            deseneaza(m.x, m.y, m.z);
        }
    }
}

void setup() {
    Serial.begin(115200);
    coada = xQueueCreate(10, sizeof(Masuratoare)); // 10 mesaje in asteptare
    if (coada == NULL) {
        Serial.println("memorie insuficienta pentru coada");
        return;
    }
    xTaskCreate(procesSenzor, "senzor", 2048, NULL, 3, NULL);
    xTaskCreate(procesEcran, "ecran", 4096, NULL, 1, NULL);
}

void loop() { vTaskDelay(pdMS_TO_TICKS(1000)); }

```

DE CE COADA ESTE MAI BUNĂ DECÂT O VARIABILĂ GLOBALĂ

- **ESTE ATOMICĂ**
 - nu puteți citi jumătate dintr-o structură scrisă pe jumătate.
- **NU PIERDE DATE**
 - o variabilă globală păstrează doar ultima valoare; coada le păstrează pe toate zece.
- **BLOCHEAZĂ EFICIENT**
 - procesul consumator doarme fără să consume procesor, în loc să verifice în buclă.
- **VĂ SPUNE CÂND SISTEMUL NU FACE FAȚĂ**
 - coada plină este un simptom măsurabil, nu o defecțiune tăcută.

8 Inversiunea de prioritate

Aceasta este cea mai renumită capcană a sistemelor în timp real. A pus în pericol misiunea **Mars Pathfinder** în 1997: roverul de pe Marte repornea singur, la intervale neregulate, iar cauza a fost exact mecanismul descris mai jos.

Cum apare

1. Un proces de **prioritate mică** ia o resursă comună (de exemplu magistrala I²C).
2. Un proces de **prioritate mare** are nevoie de aceeași resursă și se blochează, așteptând-o.
3. Un proces de **prioritate medie**, care nu are nicio legătură cu resursa, devine gata de execuție și îl întrerupe pe cel de prioritate mică.
4. Rezultat: procesul de prioritate **medie** îl întârzie, indirect, pe cel de prioritate **mare**. Ierarhia priorităților s-a inversat.

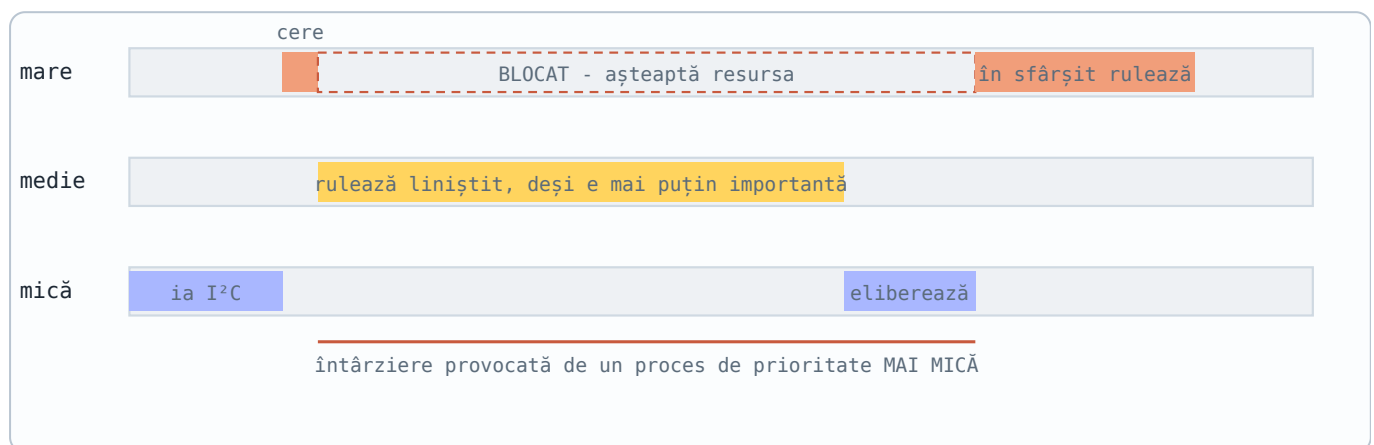


Fig. 1 - Inversiunea de prioritate. Procesul de prioritate medie nu atinge niciodată resursa disputată, dar întârzie totuși procesul cel mai important, prin simplul fapt că îl întrerupe pe deținătorul ei.

Provocați-o singuri

inversiune.ino - o singură linie face diferența

```

SemaphoreHandle_t resursa;

void asteapta_activ(uint32_t ms) {
    // Bucla de asteptare ACTIVA: tine procesorul ocupat, ca o prelucrare reala.
    uint32_t t0 = millis();
    while (millis() - t0 < ms) { }
}

void procesMic(void* p) {                                // prioritate 1
    for (;;) {
        xSemaphoreTake(resursa, portMAX_DELAY);
        Serial.println(" [mic] am luat resursa");
        asteapta_activ(300);                             // lucreaza cu resursa
        Serial.println(" [mic] eliberez resursa");
        xSemaphoreGive(resursa);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void procesMediu(void* p) {                              // prioritate 2
    vTaskDelay(pdMS_TO_TICKS(50));                       // porneste dupa cel mic
    for (;;) {
        Serial.println(" [mediu] ocup procesorul 800 ms");
        asteapta_activ(800);                             // nu atinge resursa deloc!
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void procesMare(void* p) {                              // prioritate 3
    vTaskDelay(pdMS_TO_TICKS(100));                      // cere resursa dupa cel mic
    for (;;) {
        uint32_t t0 = millis();
        Serial.println("[MARE] cer resursa");
        xSemaphoreTake(resursa, portMAX_DELAY);
        Serial.printf("[MARE] am asteptat %lu ms\n", millis() - t0);
        xSemaphoreGive(resursa);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void setup() {
    Serial.begin(115200);
    delay(500);

    // ---- SCHIMBATI DOAR ACEASTA LINIE ----
    resursa = xSemaphoreCreateBinary();                   // FARA mostenire de prioritate
    xSemaphoreGive(resursa);                             // semafoarele binare pornesc goale
    // resursa = xSemaphoreCreateMutex();                 // CU mostenire de prioritate
    // -----

    xTaskCreate(procesMic,    "mic",    2048, NULL, 1, NULL);
    xTaskCreate(procesMediu,  "mediu",  2048, NULL, 2, NULL);
    xTaskCreate(procesMare,   "mare",   2048, NULL, 3, NULL);
}

void loop() { vTaskDelay(pdMS_TO_TICKS(1000)); }

```

CE VEȚI MĂSURA

Cu `xSemaphoreCreateBinary()`, procesul de prioritate mare așteaptă în jur de **1000 MS**

: cei 300 ms în care resursa e ocupată, plus cei 800 ms în care procesul de prioritate medie îl întrerupe pe deținător. Cu `xSemaphoreCreateMutex()` așteptarea scade la aproximativ

300 MS

- doar cât durează efectiv lucrul cu resursa.

MOȘTENIREA DE PRIORITATE

Un *mutex* FreeRTOS nu este un semafor binar cu alt nume. Când un proces de prioritate mare se blochează pe un mutex, sistemul ridică

TEMPORAR

prioritatea deținătorului la nivelul celui blocat. Deținătorul nu mai poate fi întrerupt de procese intermediare, termină repede și eliberează resursa, iar prioritatea îi revine la valoarea inițială. Aceasta este soluția care a fost încărcată prin radio pe Mars Pathfinder.

Regula practică: folosiți

MUTEX

pentru excludere mutuală (protejarea unei resurse) și

SEMAFOR

pentru semnalizare (anunțarea unui eveniment). Sunt lucruri diferite, deși API-ul seamănă.

9 Compilarea firmware-ului InfiniTime

Trecem de la exemple la un firmware complet. InfiniTime are peste o sută de mii de linii de cod și rulează pe FreeRTOS cu interfața grafică LVGL.

VERIFICAȚI ÎNTÂI DOCUMENTAȚIA PROIECTULUI

Comenzile de compilare ale proiectelor open source se schimbă de la o versiune la alta. Înainte de a începe, citiți fișierul `doc/buildAndProgram.md` din depozit. Pașii de mai jos reflectă procedura obișnuită, dar depozitul este întotdeauna sursa de adevăr.

1

Descărcați codul sursă

```
$ descărcare
```

```
git clone https://github.com/InfiniTimeOrg/InfiniTime.git
cd InfiniTime
git submodule update --init --recursive
```

2

Compilați folosind imaginea Docker oficială

Este calea recomandată: aduce cu ea compilatorul ARM și pachetul de dezvoltare Nordic, deci nu trebuie să instalați nimic altceva.

```
$ compilare
```

```
docker run --rm -it \
  -v "$(pwd)":/sources \
  -u $(id -u):$(id -g) \
  infinitime/infinitime-build
```

Prima rulare durează mult, pentru că descarcă imaginea. Rezultatul apare în build/src/ - fișierul cu extensia .zip și sufixul -dfu este pachetul care se încarcă prin Bluetooth.

3

Explorați structura proiectului

```
$ orientare
```

```
ls src/systemtask/      # procesul central, care le coordoneaza pe celelalte
ls src/displayapp/      # procesul de afisare si aplicatiile
ls src/components/      # driverele: baterie, ecran, senzori, Bluetooth
grep -rn "xTaskCreate" src/ # unde se creeaza procesele
grep -rn "configTOTAL_HEAP_SIZE" src/FreeRTOSConfig.h
```

4

Alternativa fără ceas: simulatorul

InfiniSim compilează același cod al interfeței pentru calculator și îl afișează într-o fereastră, cu ecran și atingere emulate. Este calea cea mai rapidă pentru a dezvolta un cadran.

```
$ simulator
```

```
git clone https://github.com/InfiniTimeOrg/InfiniSim.git
cd InfiniSim && git submodule update --init --recursive
# urmati instructiunile din README pentru dependentele SDL2
```

Anatomia sistemului

Odată ce codul e compilat, căutați apelurile xTaskCreate. Veți găsi un tipar clar:

Proces	RoI	Cum este dimensionat
SystemTask	coordonează totul: pornire, somn, evenimente	prioritate mare, stivă generoasă
DisplayApp	desenează ecranul și tratează atingerile	cea mai mare stivă - LVGL cere memorie
stiva Bluetooth	întreține conexiunea radio	prioritate mare, termene stricte impuse de protocol
proces inactiv	rulează când nu are cine altcineva	aici se intră în starea de consum redus

PROCESUL INACTIV ESTE CEL MAI IMPORTANT PENTRU BATERIE

Când toate procesele dorm, planificatorul îi dă controlul procesului inactiv, care oprește ceasul procesorului până la următoarea întrerupere. Acest mecanism se numește

TICKLESS IDLE

. Dacă un singur proces din sistem așteaptă activ în buclă în loc să folosească `vTaskDelay`, procesul inactiv nu se mai execută niciodată, consumul crește de zece ori și bateria ține o zi în loc de o săptămână. Un singur `while(1){}` plasat greșit are acest efect.

10 Adăugarea unui proces propriu

Adăugăm în `InfiniTime` un proces care numără pașii detectați de accelerometru pe fiecare oră a zilei și îi publică printr-o coadă către interfața grafică. Este un exemplu mic, dar atinge toate mecanismele importante.

```
src/components/statistici/Statistici.h
```

```
#pragma once
#include <FreeRTOS.h>
#include <task.h>
#include <queue.h>
#include <array>
#include <cstdint>

namespace Pinetime {
    namespace Components {

        class Statistici {
        public:
            struct Raport {
                uint8_t ora;
                uint16_t pasi;
            };

            void Start(); // creeaza procesul
            void RaporteazaPasi(uint16_t pasi); // apelat din procesul senzorului
            bool CitesteRaport(Raport& iesire); // apelat din procesul de afisare

        private:
            static void Proces(void* param); // functia procesului
            void Bucla();

            TaskHandle_t descriptor = nullptr;
            QueueHandle_t coada = nullptr;
            std::array<uint16_t, 24> pe_ora {};
        };
    }
}
```

```
src/components/statistici/Statistici.cpp
```

```
#include "Statistici.h"

using namespace Pinetime::Components;

void Statistici::Start() {
```

```

// Coadă este mica: pastram doar cateva rapoarte in asteptare.
coada = xQueueCreate(4, sizeof(Raport));
configASSERT(coada != nullptr);

// Stiva de 512 de cuvinte = 2 KB. Procesul nu foloseste LVGL si nu
// are tamponari mari, deci atat ii ajunge. Verificati cu
// uxTaskGetStackHighWaterMark inainte de a considera valoarea finala.
BaseType_t rezultat = xTaskCreate(Proces, "statistici", 512, this,
                                   tskIDLE_PRIORITY + 1, &descriptor);
configASSERT(rezultat == pdPASS);
}

void Statistici::Proces(void* param) {
    static_cast<Statistici*>(param)->Bucla();
}

void Statistici::Bucla() {
    TickType_t ultima = xTaskGetTickCount();
    for (;;) {
        // Prioritate mica si perioada lunga: acest proces nu este urgent.
        // Cat dormim, procesul inactiv poate opri ceasul procesorului.
        vTaskDelayUntil(&ultima, pdMS_TO_TICKS(60000));
    }
}

void Statistici::RaporteazaPasi(uint16_t pasi) {
    Raport r { ora_curenta(), pasi };
    // Fara asteptare: daca coada e plina, pierdem un raport. Este acceptabil
    // pentru statistici, dar NU ar fi acceptabil pentru o alarma.
    xQueueSend(coada, &r, 0);
}

bool Statistici::CitesteRaport(Raport& iesire) {
    return xQueueReceive(coada, &iesire, 0) == pdTRUE;
}

```

CELE TREI DECIZII DE PROIECTARE, EXPLICIT

1. PRIORITATEA TSKIDLE_PRIORITY + 1

- cea mai mică peste procesul inactiv. Statisticile nu au termen limită; nu au voie să întârzie desenarea ecranului sau stiva Bluetooth.

2. STIVA DE 512 DE CUVINTE

- valoarea de pornire, care trebuie

VERIFICATĂ

cu uxTaskGetStackHighWaterMark. Pe un sistem cu 64 KB, fiecare kilooctet alocat degeaba este luat de la altcineva.

3. TIMP DE AȘTEPTARE ZERO LA TRIMITERE

- un proces de prioritate mare nu are voie să fie blocat de o coadă plină aparținând unuia neimportant. Pierdem raportul; nu blocăm sistemul.

11 Încărcarea pe ceas

Ceasul se actualizează prin Bluetooth, fără fire și fără programator. Bootloaderul păstrează versiunea anterioară și revine la ea automat dacă noul firmware nu confirmă că a pornit corect.

1

Încărcați ceasul complet

O actualizare întreruptă de o baterie descărcată este singurul mod realist de a strica un PineTime. Bateria trebuie să fie peste 50 %.

2

Instalați Gadgetbridge pe telefon

Din F-Droid. Este aplicația liberă care vorbește cu InfiniTime și care poate transfera fișierul de actualizare.

3

Transferați pachetul .zip pe telefon

Fișierul se află în `build/src/` și are sufixul `-dfu`. **Nu** îl dezarhivați - formatul arhivat este cel pe care îl așteaptă bootloaderul.

4

Porniți actualizarea

În Gadgetbridge: conectați ceasul, apoi `Firmware/App install` și alegeți fișierul. Transferul durează câteva minute. Nu îndepărtați telefonul de ceas în acest timp.

5

Confirmați noua versiune

După repornire, ceasul cere confirmarea că firmware-ul funcționează. Dacă nu confirmați, bootloaderul revine la versiunea anterioară la următoarea repornire - o plasă de siguranță elegantă, care merită studiată ca mecanism în sine.

ACTUALIZAREA ÎN SIGURANȚĂ, CA PRINCIPIU

Mecanismul de aici - două zone de firmware, confirmare explicită după pornire, revenire automată la eșec - este exact ce folosesc automobilele, routerile și sateliții. Un dispozitiv la care nu ai acces fizic

NU ARE VOIE

să poată fi transformat într-o cărămidă printr-o actualizare greșită. Citiți codul bootloaderului: este scurt și instructiv.

DACĂ CEVA MERGE PROST

Ceasul sigilat nu se poate reprograma prin fire fără a-l deschide. Bootloaderul este însă proiectat exact pentru această situație și, în practică, revenirea la versiunea anterioară funcționează. Nu întrerupeți niciodată un transfer în curs.

12 Sarcini de lucru

1. Folosind simulatorul de la secțiunea 5, determinați utilizarea maximă la care setul de trei procese rămâne planificabil sub RM. Comparați cu limita teoretică și explicați diferența.

2. Găsiți o configurație la care RM ratează termene iar EDF nu. Copiați ambele diagrame în referat și explicați exact ce decizie diferită ia EDF.
3. Încărcați `procese_baza.ino` pe ESP32-C6. Modificați `procesEcran` să dureze 250 ms (mai mult decât perioada lui) și descrieți ce se întâmplă cu celelalte procese.
4. Adăugați `procesDiagnostic` și notați stiva rămasă pentru fiecare proces. Reduceți alocările până la limita sigură și calculați câtă memorie ați recuperat.
5. Rulați `inversiune.ino` în ambele variante. Notați timpul de așteptare al procesului de prioritate mare în fiecare caz și explicați diferența prin mecanismul de moștenire a priorității.
6. Compilați `InfiniTime`. Găsiți toate apelurile `xTaskCreate` și realizați un tabel cu procesele sistemului: nume, prioritate, dimensiunea stivei și rolul fiecăruia.
7. Căutați în `FreeRTOSConfig.h` opțiunea `configUSE_TICKLESS_IDLE` și explicați, pe baza codului, ce se întâmplă atunci când toate procesele dorm.

13 Provocare

CADRANUL PROPRIU

Creați un cadran nou pentru `InfiniTime`, care afișează ora, nivelul bateriei și numărul de pași. Dezvoltați-l în simulator, apoi încărcați-l pe ceas.

Partea interesantă nu este desenul, ci

BUGETUL DE REÎMPROSPĂTARE

Fiecare redesenare a ecranului trezește procesorul și costă energie. Măsurați, cu `uxTaskGetStackHighWaterMark` și cu numărătorul de treziri:

- De câte ori pe minut se redesenează cadranul dumneavoastră?
- Care este numărul **MINIM** de redesenări necesar ca să afișeze ora corectă?
- Cu cât se prelungește autonomia dacă redesenati doar cifra care s-a schimbat, în loc de tot ecranul?

Un cadran care redesenează tot ecranul de zece ori pe secundă arată identic cu unul care o face o dată pe minut - dar consumă de sute de ori mai mult. Aceasta este, în esență, întreaga artă a programării pentru dispozitive portabile.

14 Întrebări de verificare

Care este limita de utilizare garantată de Rate Monotonic pentru trei procese?

100 %

78 %

69 %

50 %

Utilizarea totală este de 85 %, peste limita RM. Ce concluzie putem trage?

Sistemul sigur ratează termene

Testul de utilizare nu poate decide; trebuie verificat prin simulare sau analiza timpului de răspuns

Sistemul sigur funcționează

Trebuie folosit obligatoriu EDF

Ce face diferit vTaskDelayUntil față de vTaskDelay?

Este mai rapid

Menține o perioadă exactă, raportându-se la momentul precedent de lansare

Nu blochează procesul

Funcționează doar cu prioritate mare

În ce constă inversiunea de prioritate?

Planificatorul schimbă din greșală prioritățile

Un proces de prioritate medie întârzie indirect un proces de prioritate mare, întrerupându-l pe deținătorul unei resurse comune

Două procese cer aceeași prioritate

Procesul inactiv rulează prea des

Cum rezolvă un mutex FreeRTOS inversiunea de prioritate?

Interzice preempțiunea

Ridică temporar prioritatea deținătorului la nivelul procesului blocat

Dezactivează întreruperile

Redistribue prioritățile tuturor proceselor

De ce este important procesul inactiv pentru autonomia bateriei?

Rulează cu prioritate maximă

În el se oprește ceasul procesorului până la următoarea întrerupere (tickless idle)

Salvează datele pe memoria flash

Reduce frecvența magistralei

Ce întoarce uxTaskGetStackHighWaterMark?

Memoria totală alocată procesului

Numărul minim de cuvinte rămase libere din stivă de la pornire

Numărul de comutări de context

Timpul de execuție cumulat

Când folosim mutex și când semafor?

Sunt interschimbabile

Mutex pentru protejarea unei resurse, semafor pentru semnalizarea unui eveniment

Mutex doar în întreruperi

Semafor doar pentru procese de prioritate mare

Răspunsuri: 1-b, 2-b, 3-b, 4-b, 5-b, 6-b, 7-b, 8-b

1. Limita este $n \cdot (2^{1/n} - 1)$. Pentru $n = 3$ rezultă $3 \cdot (2^{1/3} - 1) \approx 0,780$, adică 78 %. Pentru n foarte mare, limita tinde la $\ln 2 \approx 69$ %.
2. Testul de utilizare este suficient, nu necesar. Multe seturi de procese cu utilizare peste limită sunt totuși planificabile - de aceea simulatorul afișează în acest caz un verdict de tip „nu se poate decide”.
3. `vTaskDelay` așteaptă un interval măsurat din momentul apelului, deci durata lucrului se adaugă la perioadă și eroarea se acumulează. `vTaskDelayUntil` se raportează la momentul precedent și păstrează perioada constantă.
4. Procesul de prioritate mare așteaptă o resursă ținută de unul de prioritate mică, iar acesta din urmă este întrerupt de unul de prioritate medie care nu are nicio legătură cu resursa. Este defectul care a afectat misiunea Mars Pathfinder.
5. Se numește moștenire de prioritate. Deținătorul urcă temporar la prioritatea celui blocat, deci nu mai poate fi întrerupt de procese intermediare, termină repede și eliberează resursa. Un semafor binar nu are acest mecanism.
6. Când toate procesele dorm, planificatorul dă controlul procesului inactiv, care intră într-o stare de consum redus. Un singur proces care așteaptă activ în buclă anulează complet acest mecanism.
7. Este valoarea minimă atinsă vreodată de spațiul liber din stivă. Dacă se apropie de zero, stiva este subdimensionată, iar depășirea ei provoacă blocări aparent aleatorii, greu de diagnosticat.
8. Mutexul exprimă proprietate: cine îl ia, îl și eliberează, și beneficiază de moștenirea priorității. Semaforul exprimă un eveniment: poate fi dat de un proces sau de o întrerupere și luat de altul. API-ul seamănă, dar semantica este diferită.

15 Resurse

Capitolul 7 - Modele de execuție și planificare în timp real

InfiniTime - codul sursă al firmware-ului

PineTime - documentația hardware

Capitolul 8 - Sincronizare, arhitecturi de kernel și RTOS-uri

InfiniSim - simulatorul pentru calculator

Documentația FreeRTOS