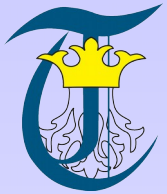

Universitatea *Transilvania* Brașov
Facultatea de Inginerie Electrică și Știința Calculatoarelor
Departamentul de Electronică și Calculatoare

Sisteme de operare distribuite și în timp real

– curs –
master SECI anul I

ș.l. dr. ing. Kertész Csaba-Zoltán

FreeRTOS

- http://freertos.org
- kernel real-time executiv
- special gândit pentru microcontrolere cu resurse limitate:
  - 6-12 kB ROM
  - 4 kB RAM
- suport pentru peste 30 de arhitecturi de microcontrolere
  - Atmel AVR, AVR32, SAM, Cypress PSoC, Infineon TriCore, Fujitsu 16FX, 32FR, Microchip PIC32, PIC24, dsPIC, NXP LPC, Renesas RX, RL78, ST STM32, TI MSP430, Xilinx Zync, MicroBlaze, PPC, ...

Real-time cu FreeRTOS

- preemptive
- planificări multiple:
  - priorități
  - Round Robin
  - cooperativ
- posibilitate de a nu dezactiva anumite întreruperi niciodată
- sincronizare (mutex, semafor binar, semafor numărător)
  - disponibil și în întreruperi

Real-time cu FreeRTOS

- temporizare software foarte eficientă
  - nu folosește procesorul sau variabile din memorie
- flaguri de evenimente non-deterministe ce nu afectează execuția întreruperilor
  - accesul la flag-uri este atomic
  - întreruperile și taskurile de timp-real funcționează cât timp flagurile trezesc taskurile de servire

FreeRTOS în sisteme încorporate

- poate rula fără tickuri (treziri periodice) în aplicații low-power
- footprint mic (6-12 kB)
- are nevoie de un singur timer, restul perifericelor pot fi accesate direct de către utilizator
- obiecte pot fi alocate atât static cât și dinamic pentru folosire deterministă și eficientă a memoriei

Dezvoltare cu FreeRTOS

- demouri pentru toate platformele suportate
- legare cu doar 3 fișiere C pentru nucleu și un fișier specific platformei
  - pentru anumite funcționalități în plus pot fi adăugate și alte fișiere sursă
- opțiuni de depanare:
  - detecția depășirii stivei
  - trace interactiv
  - urmărire explicită a taskurilor

Structura sistemului de operare

- sursele FreeRTOS pot fi descărcate sub forma unui zip care conține 2 directoare:
  - FreeRTOS: conține sistemul de operare propriu-zis
  - FreeRTOS-Plus: conține o serie de biblioteci oferind funcționalități avansate pentru anumite tipuri de aplicație:
    - interpretor de linie de comandă
    - sistem de fișiere (FAT)
    - TCP/UDP
    - trace
    - BSP
    - IoT

Structura sistemului de operare

- directorul FreeRTOS conține:
  - Source: sursele sistemului de operare
  - Demo: proiecte demo pentru toate arhitecturile și compilatoarele folosite
    - aceasta conține și niște surse comune ce implementează diferite drivere pentru periferice, comunicații pe rețea și algoritmi des întâlnite în taskuri de timp-real (abort, reset, semnalizări, polling, evenimente, etc.)

Structura sistemului de operare

- Source
  - include: toate headerele folosite (trebuie inclus în directiva de compilare)
  - portable: sursele dependente de arhitectură
    - organizat pe compilator și apoi tipul microcontrolerului
  - croutine.c: corutine (rutine cooperative)
  - event_groups.c: flaguri de evenimente
  - list.c: liste dinamice
  - queue.c: cozi de mesaje
  - tasks.c: planificare
  - timers.c: timere software

Structura sistemului de operare

- tasks.c
  - conține rutinele aparținând planificatorului:
    - creare de taskuri
    - terminare de taskuri
    - schimbare între taskuri
    - algoritmul de planificare (priorități, yield cooperativ)
    - bucla principală
    - taskul idle

Structura sistemului de operare

- queue.c
  - implementarea cozilor de mesaje
  - mutex
  - semafor binar
  - semafor de numărare
  - mutecși și semafoarele sunt defapt niște macrouri pentru cozi de mesaje specializate

Structura sistemului de operare

- list.c
  - suport pentru o listă dinamică
    - folosește listă înlănțuită
  - aceste liste sunt folosite pentru stocarea obiectelor din sistemul de operare:
    - tabela de procese (taskuri)
    - stivele
    - mesajele, sincronizări, evenimente

Structura sistemului de operare

- croutine.c
  - implementarea co-rutinelor (rutine cooperative)
  - execuție până la yield
  - soluții hibride de planificare
- event_groups.c
  - flaguri de evenimente asociate taskurilor
  - acces atomic la flaguri
  - deservirea flagurilor non-determinist

Structura sistemului de operare

- timers.c
  - timere și contoare
  - temporizarea este cuantificată la tick-ul sistemului
  - întârzieri
  - alarme

Coding style

- notația ungară (prefix cu tipul datelor)
- numele funțiilor cu camelcase și încep cu numele fișierului (după prefix)
- codul FreeRTOS este complet conform MISRA (cu două deviații bine documentate)
- indentare oarecum similară cu GNU, dar folosește tab-uri și spații goale în jurul tuturor operanzilor

Planificatorul

- se regăsește în fișierul task.c
- oferă mai multe tipuri de planificare
  - preemptivă cu priorități fixe
  - round-robin
  - cooperativă

Pornirea sistemului

```
int main()
{
  /* … Inițializări … */
  /* … creare taskuri și alte obiecte … */

  vTaskStartScheduler();
  /* … nu se ajunge niciodată aici */
  return 0; /* avoid compiler warning */
}
```

Pornirea sistemului

- vTaskStartScheduler pornește sistemul de operare, creează toate obiectele necesare, creează un task idle, după care face yield pentru orice task disponibil
- din moment ce în interiorul programului se întâmplă un context switch funcția main nu mai este reluată (cu excepția unor cazuri de eroare)
- toate buclele și funcționalitățile sistemului trebuie implementate cu task-uri

Crearea taskurilor

- xTaskCreate(pvTaskCode, pcName, usStackDepth, pvParameters, uxPriority, pxCreatedTask)
  - pvTaskCode: funcția asociată taskului
  - pcName: numele taskului (poate fi folosit la depanare)
  - usStackDepth: dimensiunea stivei (trebuie să fie suficient de mare pentru apeluri de funcții)
  - pvParameters: parametri opționali
  - uxPriority: prioritatea taskului
  - pxCreatedTask: pointer la taskul creat

Taskuri

- priorități (între 0 și configMAX_PRIORITY – 1)
- task-ul idle are prioritate 0:
  - poate avea un hook asociat
  - folosește procesorul pentru a aștepta la trezirea celorlate taskuri prin apel continuu la yield
- task-urile de aceeași prioritate pot fi rulați în Round-Robin dacă configUSE_TIME_SLICING este setat, altfel rulează până la blocare sau yield

Co-Rutine

- rutine cooperative
- asemănătoare taskurilor dar folosesc mai puține resurse pentru că ele rulează până când întâlnesc un yield
- toate co-rutinele folosesc aceleași stack

Cozi de mesaje

- sunt definite în queue.c
- permit comunicarea sincronizată între taskuri
- dacă managementul memoriei permite pot fi folosite pentru a transmite o cantitate dinamică de date
- în practică se folosesc niște obiecte specializate:
  - mutex
  - semafor binar
  - semafor de numărare

Mutex

- folosit pentru excludere mutuală
- permite moștenirea priorităților
- xSemaphoreCreateMutex()
- xSemaphoreCreateRecursiveMutex()
  - permite luarea mutexului de mai multe ori în același task

Semafor binar

- asemănător mutex-ului
- cu două stări (luat și liber)
- nu are moștenire de priorități
- folosit pentru sincronizare între taskuri și nu pentru excludere mutuală
- xSemaphoreCreateBinary

Semafor de numărare

- semafor numărător între 0 și o valoare maximă definită
- permite sincronizare la operații multiple paralele
- xSemaphoreCreateCounting

Operații pe semafoare

- xSemaphoreTake
  - operația down: decrementare atomică / blocare dacă semaforul e 0
- xSemaphoreTakeFromISR
  - variantă specială de folosit în întreruperi, nu se blochează
- xSemaphoreGive
  - operația up: incrementare atomică / trezirea taskurilor blocate
- xSemaphoreGiveFromISR
  - variantă specială pentru up ce permite sincronizarea între taskuri și întreruperi

Timeout la semafoare

- operațiile de take la semafoare permit specificarea unui timp pentru a ocupa semaforul
- dacă semaforul nu e disponibil nici după trecerea timpului definit, atunci operația eșuează și se întoarce (cu cod de eșuare
  - permite evitarea blocării indefinite a unui task critic
- dacă se dorește blocarea indefinită a semaforului atunci se poate specifica un timeout ca portMAX_DELAY (cu condiția ca opțiunea INCLUDE_vTaskSuspend să fie setată)

Mesaje generice

- câteodată poate fi mai comod transmiterea mesajelor decât sincronizarea directă a resurselor partajate
- în acest caz nu trebuie partajate deloc resurse
- xQueueCreate(uxQueueLength, uxItemSize)
  - lungimea cozii
  - dimensiunea elementelor

Transmiterea mesajului

- xQueueSendToBack(xQueue, pvItemToQueue, xTicksToWait);
- xQueueSendToFront(xQueue, pvItemToQueue, xTicksToWait);
  - xQueue: coada / stiva
  - pvItemToQueue: elementul ce trebui trimis
  - xTicksToWait: timpul de așteptare până la eliberarea cozii (indefinit cu portMAX_DELAY)
- taskul se blochează dacă nu e loc în coadă

Recepția mesajului

- xQueueReceive(xQueue, pvBuffer, xTicksToWait);
  - xQueue: coada
  - pvBuffer: zona în care să fie copiat elementul recepționat
  - xTicksToWait: perioada de așteptare până la primirea unui element
- taskul se blochează dacă coada este goală

Mesaje din întreruperi

- trebuie folosite variante dedicate:
  - xQueueSendToBackFromISR(xQueue, pvItemToQueue, *pxHigherPriorityTaskWoken);
  - xQueueSendToFrontFromISR(xQueue, pvItemToQueue, *pxHigherPriorityTaskWoken);
- parametrul pxHigherPriorityTaskWoken este setat dacă un context switch e necesar, în acest caz dezvoltatorul trebuie să se asigure că apelează context switch-ul din rutina de întrerupere

Apelarea planificatorului

- taskYIELD()
- pe anumite arhitecturi poate fi implementat separat un taskYIELD_FROM_ISR() pentru apelul din întreruperi
- poate fi apelat de rutinele cooperative pentru a ceda procesorul sau din întreruperi dacă acțiunile din rutina de servire au trezit un task mai prioritar
- în modul preemptiv este apelat automat în tick-ul sistemului sau la rutinele apelate din user space care trezesc task-uri

Blocarea / trezirea taskurilor

- blocarea implicită
  - funcțiile de sincronizare și de întârziere pot bloca taskul în mod implicit
  - poate fi trezit automat de apariția evenimentului
- blocarea explicită
  - vTaskSuspend()
  - taskul este blocat indefinit până la trezirea explicită cu vTaskResume() apelat dintr-un alt task sau vTaskResumeFromISR() apelat dintr-o rutină de servire a întreruperilor

Întârzieri

- taskurile pot fi blocate pentru o perioadă definită
- vTaskDelay(xTicksToDelay)
  - taskul este blocat pentru xTickToDelay cuante de timp
  - trezirea este garantată după trecerea a tick-urilor precizate relativ la tick-ul curent, dar datorită faptului că alte taskuri pot fi și ele rulate, această funcție nu poate fi folosită pentru taskuri repetitive de frecvență fixă

Întârzieri

- vTaskDelayUntil(pxPreviousWakeTime, xTimeIncrement)
  - întârziere până la un moment de timp absolut
  - permite realizare unor taskuri repetitive de frecvență fixă
  - pxPreviousWakeTime va conține momentul de timp când a fost trezit taskul, chiar dacă din cauza altor taskuri execuția se întâmplă mai târziu

Oprirea planificatorul

- în modul preemptiv planificatorul poate fi oprit temporar pentru a evita înlocuirea unor secțiuni critice dintr-un task
- vTaskSuspendAll()
- vTaskResumeAll()
- toate context switchurile sunt oprite pe perioada suspendării
- apeluri de funcții blocante care cauzează context switch în mod normal (vTaskDelayUntil, xQueueSend, etc...) nu trebuie apelate pe parcursul suspendării

Configurarea sistemului

- pe lângă legarea fișierelor cod sursă a sistemului de operare, utilizatorul este obligat să creeze și un fișier numit FreeRTOSConfig.h
- în acest fișier trebuie setate toate flagurile care controlează sistemul de operare:
  - flaguri de configurare
    - de ex. configUSE_PREEMPTION
  - flaguri de includere a componentelor
    - de ex. INCLUDE_vTaskSuspend
  - toate configurațiile posibile sunt descrise la http://www.freertos.org/a00 110.html