# 1-2. Tétel

1. Mutassa be a uCOS-II felépítését és főbb szolgáltatásait

2. Ismertesse a microOS-II-ben a taszkok nyilvántartásához használt struktúrákat és használatukat! Mutassa be, hogy a nyilvántartásban hogyan lehet egy taszkot futásra késszé tenni, törölni a futásra készek közül, valamint megtalálni a legnagyobb prioritású futásra kész taszkot!

## microC/OS-II

* általános célú RT OS
* egyszerű és icipici
* jól konfigurálható 🡪 jól optimalizálható az erőforrás igénye
* C nyelven írt forráskód elérhető
* nem open source, üzelit célokra meg kell venni
* széles CPU támogatás
* jó commmunity

## OS felépítése

16 fájl

### Processzorfüggő fájlok – Port files

* minimális platformfüggő rész – interfész a platformfüggetlen rész és a HW között
* fájlok:
  + OS\_CPPU.H
    - proc specifikus definíciók
    - adattípus definíciók
      * pl. nem engedhető meg, hogy az int minden processzoron más legyen (nem kiszámítható memórahasználat pl.)
      * INT8U, INT8S, INT16U, INT16S, INT32U, INT32S, FP32 stb…
    - kritikus szakaszokat kezelő makrók
      * OS\_ENTER\_CRITICAL()
      * OS\_EXIT\_CRITICAL()
  + OS\_CPU\_C.C
    - implementációs source
    - HW időzítők
    - megszakítás kezelők
    - kontextus váltó függvények
    - task kezelő fv-ek
    - néhány extra fv. (az OS kódját tilos változtatni, ezért néhány dolgot itt kell implementálni)
  + OC\_CPU\_A.ASM
    - builded code
* sok platformra meg vannak írva, de ha nincs a miénkre, nekünk kell implementálni

### Processzorfüggetlen fájlok

* a legtöbb funkció itt van
* fájlok:
  + UCOS\_II.H, UCOS\_II.C
    - OS szolgáltatásainak függvényei (hogy akarjuk-e őket használni, az a konfig fájlokban állítható)
  + OS\_CORE.C
    - a legalapvetőbb függvények implementációja
  + OS\_TASK.C
    - scheduler
    - task kezelő függvények
  + OS\_TIME.C
    - időzítés
  + OS\_FLAG.C – flagek implementációja
  + OS\_SEM.C – semaphore
  + OS\_MUTEX.C – mutexek
  + OS\_MBOX.C – msg box kezelés
  + OS\_Q.C – queue
  + OS\_MEM.C – memória management

### Konfigurációs fájlok

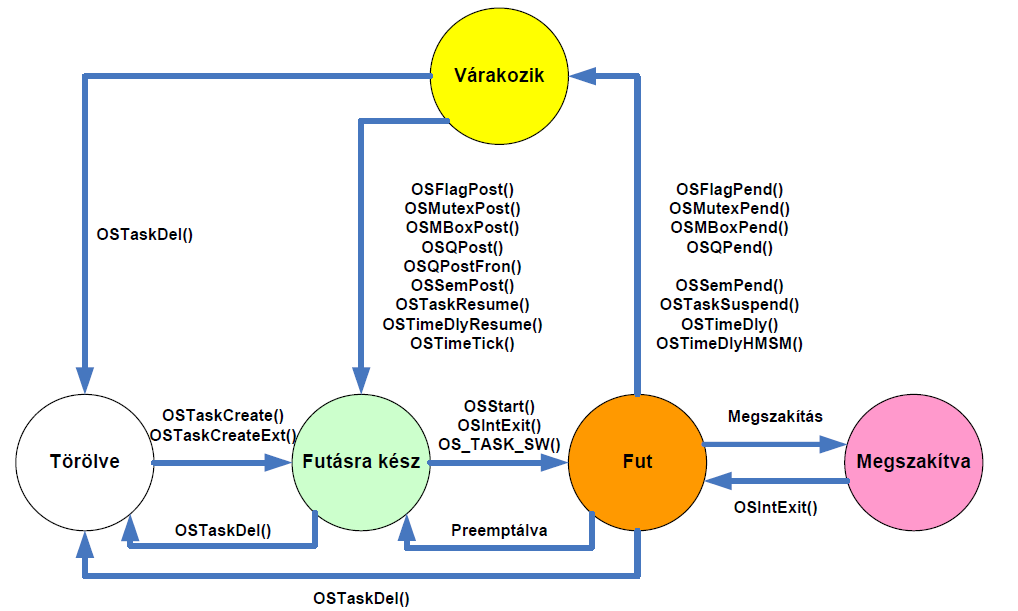
* master include fájl: INCLUDES.H
  + az OS minden sourcánál be van includeolva
  + globális hivatkozások, változók
  + regiszterek definíciója
  + könyvtári függvények leírói
* fő config: OS\_CFG.H
  + alapszolgáltatások (pl. taskok kezelése nem kapcsolható ki)
  + összes kiegészítő szolgáltatás ki és be kapcsolható
  + tiltás lehet
    - globális szolgáltatás tiltás
    - egyes funkciók/függvények tiltása

## Konfigurálás

* alapszabályok
  + minimális RAM igény
  + minimális kódmemória igény
  + az OS-re épülő alkalmazásnak is kell hagyni erőforrást
* néhány beállítás
  + OS\_MAX\_EVENTS
  + OS\_MAX\_FLAGS
  + OS\_MAX\_QS
  + OS\_MAX\_TASKS
  + OS\_LOWEST\_PRI
  + nem csak funkciókat, hanem azok egyes részeit is egyben lehet állítani
    - flagek
    - mutexek
    - semaphoreok
    - mailboxok
    - queuek
  + memória menedzsment ENABLE/DISABLE
  + memória menedzsment query enable/disable

## Taszkok

### Állapotok



### Taskok kezelése

* create: INT8U OSTaskCreate(void (\*task) (void \*pd), void \*p\_data, OS\_STK \*ptos, INT8U prio)
  + ütemezőnél történő beregisztrálás
  + \*pd – task kódját tartalmazó fv. címe
  + \*p\_data – fv. argumentumai
  + prio **– prioritás = a task azonosítója**
  + task-fv. prototípusa: void TaskFunction(void \*pd);
* delete: INT8U OSTaskDel(INT8U prio);
  + a prioritásnak nem feltétlenül kell fixnek lennie egész élete során
* prioritás változtatása: INT8U OSTaskChangePrio(INT7U oldprio, INT8U newprio);
* task felfüggesztése, visszaállítása (ha pl. vezérlőpultról egy funkciót tiltanak, vagy engedélyeznek)
  + INT8U OSTaskSuspend(INT8U prio);
  + INT8U OSTaskResunme(INT8U prio);
* annyi task lehet, amennyi prioritást engedélyezünk
* taskok nyilvántartása: Task Control Block – TCB
  + a beregisztrált taszk pointereit tárolja
* egy task paraméterei
  + várakozó állapot? ha igen, tároljuk melyik objektumra várunk
  + ha várakozásban van, tárolva van, mennyi systick-ig kell még várni, amikor ez nulla, a task futásra kész
  + taszk státusza (fut, várakozik)
  + prioritása
  + törölhető-e a task

### Ütemezés

* nagyon kritikus, hogy gyors legyen, és hatékony
* ütemezéshez: taszkok prioritásán alapuló nyilvántartási rendszer: azt rögzíti, hol van futásra kész task 🡪 néhány utasítással
  + be lehet rögzíteni új prioritást
  + el lehet távolítani prioritást
  + meg lehet keresni a legmagasabb prioritásút
* legyen 64 prioritási szintünk 🡪 1 prioritást 6 biten tudunk ábrázolni
* az OS-II-ben létrejön egy 8x8-as mátrix a prioritások nyilvántartására (8 db 1byteos vektor) – OsrdyTbl
  + a prioritás felső 3 bitje a sorokat fogja jelölni
  + az alsó 3 bit az oszlopokat
* nyilván tartja mely sorokban van futásra kész task 🡪 minden sorhoz egy elem 🡪 OSrdyGrp vektor
* összesen 9 byte
* segédtábla a taskok futásra késszé tételéhez, illetve eltávolításához a táblázatból:
  + OSMapTbl – egy olyan 8 elemű vektorokból összeálló tömb, ahol az adott indexhez tartozó vektornak az annyiadik eleme 1, ahanyadik index
* task futásra késszé tétele: az OSRdyTbl-ben a megfelelő elemet 1-re kell írni, az OSRdyGrp-ben a megfelelő elemet 1-esre kell írni, ha még nem az (legyen a prio: 12: 0x001 010b)
  + OSRdyGrp |= OSMapTbl[prio>>3]; – veszi a prioritás felső 3 bitjét (a táblázatban a sornak a számát), és az ennyiedik vektort az OSMapTbl-ből összevagyolja az OSRdyGrp-vel
    - OSMapTbl[prio>>3] – OSMapTbl[001] – 0x0000 0010b
  + OSRdyTbl[prio>>3] |= OSMapTbl[prio & 0x07h] – kimaszkolja a prio felső 3 bitjét, a maradék alapján veszi az annyiadik elemét, az OSMapTbl-nek, ahanyadik oszlopban van a prio 🡪 összevagyolja az OSRdyTbl megfelelő vektorával
* task eltávolítása a futásra készek közül
  + if ((OSRdyTbl[prio>>3] &= ~OSMapTbl[prio & 0x07h]) == 0x00h
    - OSRdyGrp &= ~OSMabTbl[prio>>3];
  + maszkolja a prio felső hármasát 🡪 ennyiedik elemét veszi az OSMapTbl-nek 🡪 veszi a negáltját 🡪 összeéseli az OSRdyTbl megfelelő vektorával 🡪 ha az eredmény pontosan 0 🡪 törli a megfelelő elemet az OSRdyGrp-ből
* legmagasabb prioritású futásra kész task megtalálása:
  + segédtábla: OSUnMapTbl[] = { …} – speckó tábla, ami segít megadni, hogy egy vektorban hanyadik a legkisebb helyiértékű valid bit (a sorszám alapján)
  + Y = OSInMapTbl[OSRdyGrp]; - megmondja melyik a legkisebb helyiértékű valid bit a RdyGroupban – megmondja melyik sorban van a legnagyobb prioritású task
  + X = OSUnMapTbl[OSRdyTbl[Y]]; - amelyik sorban van a legkisebb helyiértékű valid bit a RdyTbl-ben, az ehhez tartozó sorban melyik a legkisebb helyiértékű valid bit
  + prio = (Y<<3) + X; - felső 3 bit + alsó 3 bit összerakása
* scheduler működése
  + megszakítások tiltása
  + gyorsan kikeressük a legnagyobb prioritású futásra kész taszkot az előzőekkel
  + amennyiben a futónál elvégezzük a taszkváltást, ha nem fut tovább a jelenlegi
  + engedélyezzük a megszakításokat

## Megszakítások

* az OS-nek tudnia kell, hogy interruptban vagyunk 🡪 különben megszakítódhat
* külön fv. erre: OSIntEnter() és OSIntExit()

## OS indulása

* többlépéses folyamat
  + OSInit();
  + OSTaskCreate(); // legalább egy taskot létre kell hozni
  + OSStart();

## Időzítés

* system tick késleltetés: void OsTimeDly(INT16U tick)
  + nem pontos időzítés 🡪 a megadott idő letelte után csak akkor fut futni, ha ez lesz a legmagasabb futásra kész task 🡪 csak a késeltetés minimum idejét tudjuk megadni, amin belül biztosan nem fut
* késleltetés SI mértékegységgel: INT8U OSTimeDlyHMSM(INT8U hours, INT8U minutes, INT8U seconds, INT16U milli);
* máshonnan feloldani a várakozásban levő feladatot: INT8U OSTimeDlyResume(INT8U prio);
* indítás óta eltelt idő: INT32U OSTimeGet(void);
* OS idejének beállítása: void OSTimeSet(INT32U ticks);

## Szinkronizációs objektumok

Konfiguráció: ugyanolyan struktúrával (OS\_EVENT)

* OSEventType – objektum típusa
* OSEventTbl[OS\_EVENT\_TBL\_SIZE] - az ütemezéshez hasonlóan az adott objektumra várakozó taszkok prioritásainak tárolása
* OSEventGrp – az OSEventTbl-hez tartozó Grp vektor
* OSEventCnt – a szemaforok esetén a szemafor számlálója
* \*OSEventPtr – mail-boxes és msg queue-k esetén használatos – a bepakolt dolgok pointerét tartalmazza

### Szemaforok

* létrehozása: OS\_EVENT \*OSSemCreate(INT16U cnt);
  + cnt – szemafor számlálójának értéke
* törlése: OS\_EVENT \*OSSemDel(OS\_EVENT \*pevent, INT8U opt, INT8U \*err);
  + \*pevent – melyik szemafort akarjuk törölni
  + törölhető-e a szemafor, ha valaki várakozik rá
* bejelentkezés a várakozási sorba: void OSSemPend(OS\_EVENT \*pevent, INT16U timeout, INT8U \*err);
* értékének vizsgálata: INT16U OSSemAccept(OS\_EVENT \*pevent);
* felszabadítás: INT8U OSSemPost(OS\_EVENT \*pevent);

### Egyéb kommunikációs eszközök

* **Mutex** – prioritás inverzió ellen prioritás öröklést tartalmaz
* **Eseményjelző flag**
* **Postaláda**
* **Várakozási sor**

## Taszkok használata

* taszkszervezési megoldások
  + egyszeres lefutású (single-shot)
    - lefut 🡪 szól az OS-nek, hogy done 🡪 törli önmagát
    - általában init feladatokra
  + végtelen ciklusú
    - általában ezt használjuk
    - célszerű olyan fv-hívásokat implementálni, amelyek várakozó állapotba viszik a taszkot 🡪 ha egy magas prioritású sosem kerül várakozásba, az alacsonyak kiéheznek
* végtelen ciklusú task implementálása
  + inicializáló task (magas prioritású)
    - HW init
    - létrehozza a szinkronizációs objektumokat
    - létrehozza a végtelen ciklusú taszkot
    - törli magát
  + main task
    - while(1)
    - megcsinálja a dolgát
    - elmegy aludni egy időre