Arhitecturi software pentru sisteme embedded

Componente de bază. Tipuri de arhitecturi.

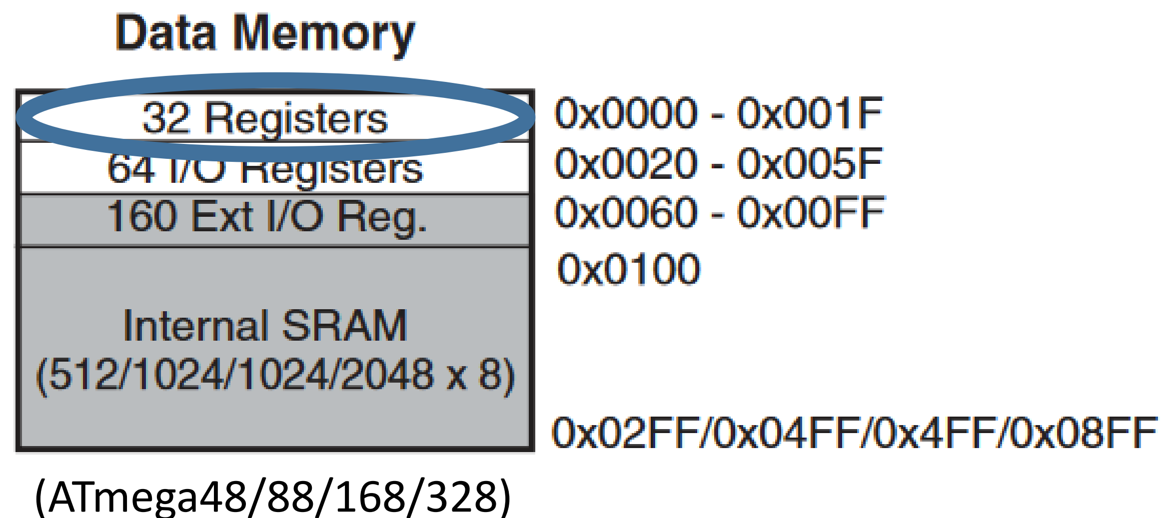
Componente de bază

- Fișiere header – pentru regiștrii perifericelor
- Drivere – pentru periferice
- Stivă comunicare – pentru protocoalele de comunicare utilizate
- Cod startup – pregătește sistemul pentru execuția programului
- Funcționalitate reprogramare – necesară pentru rescrierea firmware-ului

Fișiere header

Nu uitați:

Regiștrii sunt mapați în zona de memorie adresabilă de către procesor.



(ATmega48/88/168/328)

Fișiere header

Fără definirea și utilizarea unor fișiere header codul ar arăta așa:

```
...
/* Seteaza valori port B*/
*((uint16 *) 0x0005) =
(1<<7)|(1<<6)|(1<<1)|(1<<0);
/* Seteaza directia pentru pini */
*((uint16 *) 0x0004) =
(1<<3)|(1<<2)|(1<<1)|(1<<0);
/* Citeste valori pini*/
i = *((uint16 *) 0x0005);
...
```

În loc să arate arăte așa:

```
...
/* Seteaza valori port B*/
PORTB =
(1<<PB7)|(1<<PB6)|(1<<PB1)|(1<<PB0);
/* Seteaza directia pentru pini */
DDRB =
(1<<DDB3)|(1<<DDB2)|(1<<DDB1)|(1<<DDB0);
/* Citeste valori pini*/
i = PINB;
...
```

Universitatea Politehnica Timișoara

Fișiere header

- Fișierele header definesc simboluri pentru regiștrii și componentele acestora
- Utilizarea lor ușurează munca programatorului și permite scrierea de cod intuitiv

```
#define DDRB _SFR_IO8(0x04)
#define DDB0 0
#define DDB1 1
#define DDB2 2
#define DDB3 3
#define DDB4 4
#define DDB5 5
#define DDB6 6
#define DDB7 7

#define PORTB _SFR_IO8(0x05)
#define PORTB0 0
#define PORTB1 1
#define PORTB2 2
#define PORTB3 3
#define PORTB4 4
#define PORTB5 5
#define PORTB6 6
#define PORTB7 7
```

(avr-libc/avr-libc/include/avr/iom328p.h)

Drivere

- Se folosesc direct de regiștrii modulului pentru a implementa o funcționalitate specifică
- Componente ce oferă o interfață software pentru un modul hardware (intern sau extern)
- Oferă utilizatorului un nivel de abstractizare simplificat

Drivere – Exemplu AVR sleep management

Interfața permite utilizarea facilă a funcției de management al consumului de putere.

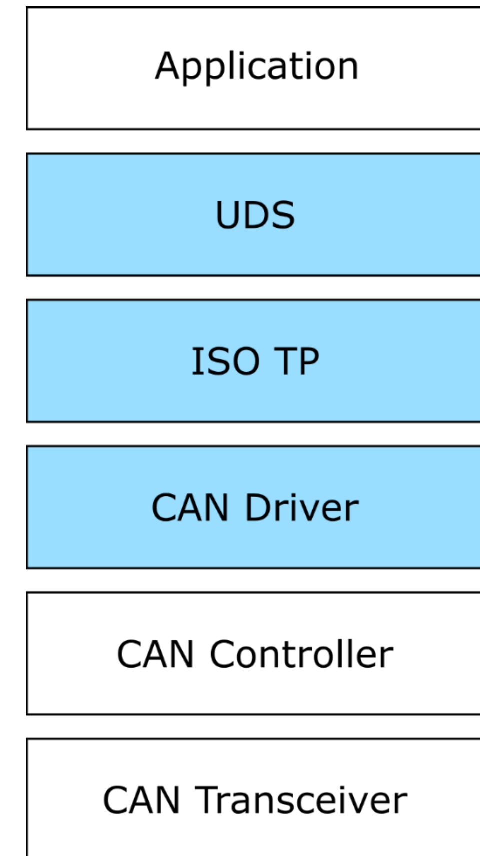
```
void    sleep_enable (void);
void    sleep_disable (void);
void    sleep_cpu (void);
void    sleep_mode (void);
void    sleep_bod_disable (void);
```

Stivă comunicare

- Implementare în layere a unui protocol de comunicare
- La nivelul inferior avem driverul care interacționează direct cu modulul hardware responsabil cu transmiterea înspre sau recepționarea dinspre nivelul fizic
- Nivelele superioare sunt reprezentate de layerele definite prin specificația protocolului
- Fiecare nivel oferă un grad de abstractizare necesar nivelului superior

Stivă comunicare – Exemplu stivă diagnoză CAN

- Stiva utilizată în aplicații de diagnoză având CAN (Controller Area Network) ca suport fizic:
  - Driverul interfațează layerul fizic
  - ISO TP – protcocolul de transfer = împachetarea informațiilor
  - UDS – Unified Diagnostic Services = specifică tipurile de mesaje specifice pentru de diagnoză



Universitatea Politehnica Timișoara

Cod startup

- Execută operații inițiale menite să aducă sistemul în starea premergătoare execuției programului principal
- Nu este introdus automat de către procesor
- De regulă se generează automat cu funcționalități minimale la crearea unui nou proiect
- Programatorul trebuie să se asigure că există și ca efectuează operațiunile așteptate

Cod startup – operații

- Copiază date inițializate și/sau cod din ROM în RAM
- Scrie “0” în zona de date neinițializată
- Alocă spațiu pentru stivă și o inițializează
- Inițializează registrul stack pointer al procesorului
- Crează și inițializează zona heap (dacă este necesar)
- Execută constructorii pentru toate variabilele globale (C++)
- Apelează funcția `main()`

Universitatea Politehnica Timișoara

Reprogramare

- În multe cazuri este necesară reprogramarea dispozitivelor embedded
- Pentru reprogramarea dispozitivului este necesar ca funcționalitatea de reprogramare să fie deservită de software-ul existent pe dispozitiv
- Componenta responsabilă pentru reprogramare nu este activă în mod normal. Activarea ei în fiecare caz se face prin metode specifice

Cum abordăm operațiunea de reprogramare?

- Soft-ul curent trebuie înlocuit. Cum facem înlocuirea?
- Cum înlocuim modulul responsabil cu reprogramarea (flashloader)?
- Sistemul trebuie să fie utilizabil/reprogramabil indiferent de rezultatul încercării de reprogramare

Exemplu de abordare greșită:

1. Ștergere zonă flashloader
2. Programare flashloader nou în aceeași zonă

În cazul întreruperii alimentării sistemul pierde posibilitatea de reprogramare! → Brick

Reprogramarea flashloader-ului

1. Flashloader-ul vechi nu este șters
2. Cel nou se programează într-o zonă diferită de memorie rezervată pentru acest scop
3. După programarea cu succes a versiunii noi se marchează ca flashloader valid cel din zona nou programată și se invalidează cel vechi

Pasul 3 se poate îndeplini în două moduri în funcție de suportul hardware:

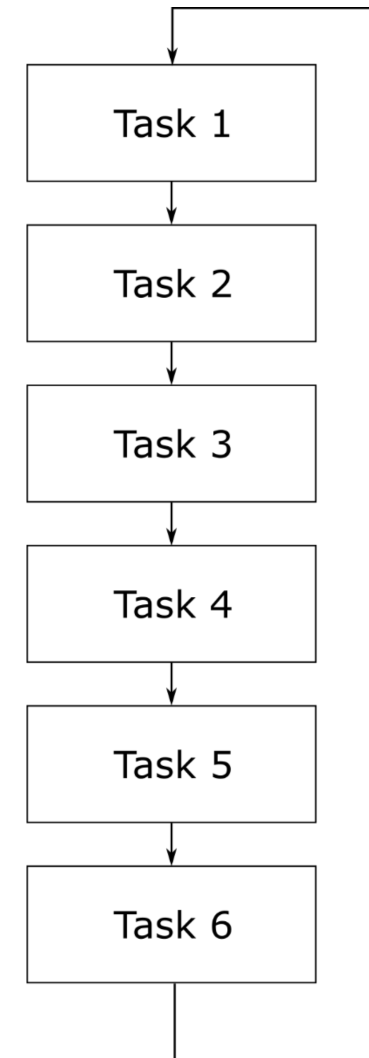
- Prin funcționalitatea de bootswap – adresele celor 2 zone de memorie sunt interschimbate printr-o funcționalitate hardware a chipului
- Prin implementarea unui mecanism software ce folosește un flag păstrat într-o memorie nonvolatilă pentru identificarea zonei valide

Arhitecturi SW de bază în sisteme embedded

- Round robin ← Cea mai simplă formă
- Round robin cu întreruperi
- Scheduling prin coadă de funcții
- Sistem de operare în timp real ← Cea mai complexă formă

Round robin

- O buclă principală în care fiecare componentă este deservită pe rând
- Avantaje:
  - Cea mai simplă arhitectură
  - Fără întreruperi
  - Nu este afectată de problema partajării datelor
- Dezavantaje:
  - Perioada de execuție a fiecărei componente este limitată de durata fiecărei bucle
  - Dacă execuția unei componente este prelungită pot să apară probleme la nivelul celorlalte componente
  - Extinderea cu funcționalități adiționale poate afecta funcționalitatea



Round robin – exemplu cod

```
void main(void) {
        while(TRUE) {
                if (device_A needs service)
                        service device_A
                if (device_B needs service)
                        service device_B
                if (device_C needs service)
                        service device_C

                ...
                if (device_N needs service)
                        service device_N
        }
}
```

Round robin – exemplu cod îmbunătățire perioadă execuție

```
void main(void) {
        while(TRUE) {
                if (device_A needs service)
                        service device_A
                if (device_B needs service)
                        service device_B
                if (device_A needs service)
                        service device_A
                if (device_C needs service)
                        service device_C
                ...
                if (device_N needs service)
                        service device_N
        }
}
```

Round robin cu întreruperi

- Sarcinile urgente sunt efectuate prin rutine de tratare a întreruperilor
- În lipsa unor sarcini urgente bucla principală funcționează fără întreruperi

- Avantaje:
  - Timp de răspuns îmbunătățit pentru sarcini prioritare
  - Relativ simplu de implementat
- Dezavantaje:
  - Poate să apară problema datelor partajate
  - Timpul de răspuns pentru sarcini mai puțin urgente nu e constant

Round robin cu întreruperi – exemplu cod

```
/* Flag for device_A follow-up processing */
BOOL flag_A = FALSE;

/* Interrupt Service Routine for device_A */
ISR_A(void) {
        /* handle urgent requirements for
        device_A in the ISR */
        /* set flag for follow-up processing in
        the main loop */
        Flag_A = TRUE;
}

void main(void) {
while(TRUE) {
        if (flag_A) {
                flag_A = FALSE
                /* Do follow-up
                processing with data
                from device_A */
        }
if (device_B requires service)
        service device_B
if (device_C requires service)
        service device_C
...
if (device_N requires service)
        service device_N
}
}
```

Scheduling prin coadă de funcții

- Rutinele de tratare a întreruperilor execută sarcini urgente și adaugă într-o coadă pointeri către rutinele ce urmează să fie executate în bucla principală
- Adăugarea în coadă nu trebuie să fie neapărat la sfârșitul cozii. Se poate lua în considerare și adăugarea în funcția priorității de execuție
- Bucla principală va executa rutinele în ordinea în care acestea se găsesc în coadă

Scheduling prin coadă de funcții

- Avantaje:
  - Latența pentru rutinele cu prioritate mare se redusă față de varianta round robin cu întreruperi
  - Ordinea execuției rutinelor modificată dinamic adaptată la necesități
- Dezavantaje:
  - Poate crește latența pentru rutinele cu prioritate scăzută
  - Este posibil ca rutine cu prioritate foarte scăzută să nu fie executate niciodată
  - Algoritmul de introducere în coadă poate să fie complex și costisitor ca timp de execuție

Scheduling prin coadă de funcții – exemplu cod

```
void taskA(void) {
    //Non-urgent operation for A
}
void taskB(void){
        //Non-urgent operation for B
}
/* Interrupt Service Routine for device_A */
ISR_A(void) {
        /* Handle urgent requirements for
        device_A in the ISR */
        /* Add task B to queue */
}
```

```
/* Interrupt Service Routine for device_B */
ISR_B(void) {
        /* handle urgent requirements for
        device_B in the ISR */
        /* Add task B to queue */
}
void main(void) {
        while(TRUE) {
                while(queue_not_empty){
                        /*Get task from queue*/
                        /*Execute task*/
                }
        }
}
```

Sistem de operare în timp real

- Rutinele de tratare a întreruperilor execută sarcini urgente și semnalează prezența unor sarcini mai puțin urgente
- Sistemul de operare invocă în mod dinamic sarcinile non-urgente
- Sistemul de operare poate să suspende execuția unei rutine mai puțin urgente în favoarea alteia mai prioritare
- Comunicare între rutine este realizată prin intermediul sistemului de operare
- Elementele de bază ale unui sistem de operare sunt task-urile.

Sistem de operare în timp real

- Avantaje:
  - Prin suspendarea task-urilor se reduce timpul de așteptare al task-urilor prioritare la 0
  - Mecanismele de scheduling crează un sistem stabil ca timp de răspuns
- Dezavantaje:
  - Cost și complexitate ridicate
  - Complexitatea duce la consum suplimentar de timp

Tipuri de task-uri

- Preemptive – pot fi întrerupte oricând pentru execuția unor task-uri mai prioritare
- Non-preemptive – nu pot fi întrerupte indiferent de prioritatea task-urilor ce așteaptă să fie executate

Exemple de sisteme de operare în timp real

- FreeRTOS – Foarte popular pentru o serie mare de aplicații
- Integrity – Sistem de operare performant utilizat în aplicații safety-critical (aeronautică, automotive)
- OSEK – utilizat în domeniul automotive
- TI-RTOS – oferit de Texas Instruments în general pentru platformele proprii
- TizenRT – dedicat pentru dispozitive IoT

Alegerea arhitecturii potrivite

- Alegeți arhitectura cea mai simplă care se potrivește cu cerințele
- Dacă timpii de răspuns indică necesitatea unui sistem de operare alegeți de la început această variantă. O migrare ulterioară la o arhitectură mai complexă se va dovedi mai complexă
- Dacă este cazul se pot folosi abordări hibride