**CDL – Arhitectura software**

**Intro: (could be skipped)**

Vom explica arhitectura la nivel de **fisiere, module si functii** si **cum se mapeaza cerintele pe cod. (mai detaliat, explicand de ce am scris codul asa si nu altfel).** Document Word nu PPT fiindca nu imi formateaza codul bine, dar o sa fie suficiente snippeturi de cod si poze, nu doar text.

Ca sa aiba sens organizarea proiectului final vom prezenta ocazional codul prin comparatie cu 2 variante intermediare:

1.**Nefactorizata (**prima varianta pe care am scris-o si mergea in simulare): tot codul in main\_task, mai deloc abstractii, fara responsabilitati clare, entanglement (ex: unde citesc butoanele schimb si modul si mai iau si amprente de timp, aprind si leduri), stare comuna pe program: am niste variabile statice in main\_task pe care poate sa mi le modifice oricine

**2.Prima factorizare:** Organizare pe fisiere. Cateva abstractii care ma ajuta: struct pentru **actuatori, leduri, timer** etc. si functii care opereaza pe ele. Functii pentru executia fiecarui mod. Am mai taiat din variabilele statice.

Si varianta finala care foloseste state pattern. Explicat teoretic (din carti) si ce rezolva la problema noastra.

**Obs:** Nu folosim nimic din **Arduino.h**, doar setup() si loop() in loc de main() si while(1) fiindca altfel nu merge sa simulam. Desi toate fisierele sunt cpp, folosim doar biblioteca pt. LCD care necesita C++, toate fisierele in afara de lcd.cpp sunt scrise in plain **C**.

A screenshot of a computer program

AI-generated content may be incorrect.**Structura de fisiere: (could be skipped)**

Pornim de la structura default a unui proiect facut in PlatformIO.

El cauta si fisierele sursa in **src** si in headerele in **include.** (se pot

pune headerele si in src).

Avem in **src**:

**main.cpp:** **setup()** si **loop(),** in setup: **securitate**,

in loop: rulam **main\_task()** o data la 20ms

**main\_task.cpp:** **updateaza stareabutonulu**i,

**afiseaza pe lcd** modul curent si butonul si **ruleaza o iteratie**

**a fsm-ului**

In **src/utils**: avem abstractii utile in cod, separam totodata

responsabilitatile:

**buttons.cpp**: debounce; detect press, release la butoane.

**kbd.cpp:** citire si debounce a unui keypad 4x3

**lcd.cpp:** wrapper in jurul **LiquidCrystal.h** ca sa nu depind de el

**registers.cpp:** implementare cu PORT,PIN,DDR, timere a **pinMode(),**

**digitalRead(), digitalWrite(), millis()**

**scheduler.cpp:** ca sa rulez periodic mai multe taskuri

**timer.cpp:** interfata mai frumoasa la luat timestampuri

Acum, niste functionalitati pentru lucrul mai usor cu componentele:

**actuators.cpp**: rotirea motorului

**leds.cpp**: patternurile cu leduri cerute

**lcd\_pattern.cpp:** afisarea starii si a butonului

**kbd\_security.cpp:** ultima cerinta

Partea de **FSM**: starile sunt **UNLOCKED, LOCKED, DOUBLE\_LOCKED**

**door\_states.cpp**: init, aflare stare curenta, anterioara, update la stare etc.

**[unlocked, locked, doublelocked]\_mode.cpp**: creaza starea si ii pun metodele apelate: la intrarea in stare, permanent cat esti in stare, detectie de tranzitie si la iesirea din stare

Fisierele **.h** cu **typedefurile** si **antetele** de functii din **include** au aceeasi structura de foldere ca **.cpp** din **src**.

**Simularea:** in **Simulide** si **Proteus**:

A computer screen shot of a computer

AI-generated content may be incorrect. Motoare conectate direct la pini. In viitor se pot folosi tranzistoare sau control cu PWM.

A computer screen shot of a circuit board

AI-generated content may be incorrect.

**Maparea cerintelor pe cod: aici discutam implementarea si de ce am scris codul asa, prezentam doar partea de cod mai interesanta:**

O luam de la cerintele care necesita cel mai putin cod de explicat in sus:

**2.9 REQ208: Cyclic routine**

**The main routine of the software shall be running within a 20ms task.**

Am creat un struct **scheduler** care are o lista de taskuri (ptr la functii) si le ruleaza periodic. La inifinit in **loop(),** trece prin ele si daca a trecut suficient timp de ultima data cand a rulat un task il ruleaza iar.

void

*check\_events*(*scheduler*\* *s*)

{

    for (*u8* task = 0; task < *s*->crt\_task; task++)

    {

        if (*millis\_*() - *s*->last\_exec\_times[task] >= *s*->period[task])

        {

*s*->last\_exec\_times[task] = *millis\_*();

            (*s*->tasks[task])();

        }

    }

}

**Loop()** adauga **main\_task()** in lista de taskuri:

void

*loop*(void)

{

    static *scheduler* s;

    static *bool\_* sched\_init = *FALSE*;

    if (!sched\_init)

    {

        sched\_init = *TRUE*;

        const *u8* HOW\_OFTEN\_RUN\_MAIN\_LOOP = 20;

*add\_task*(&s, *main\_task*, HOW\_OFTEN\_RUN\_MAIN\_LOOP);

    }

*check\_events*(&s);

}

**Decizie:** Se vede ca loop-ul face initializarea schedulerui, si trickul ca sa o faca o singura data este sa iei o variabila statica care retine daca ai initializat si la initializare o faci TRUE.

De ce nu initializarea in **setup()**?

Fiindca **setup()** si **loop()** nu au parametrii, deci pot comunica intre ele doar prin variabile “globale”. Initul in setup ar fi insemnat sa scot schedulerul ca static dar la nivel de fisier in afara functiei.

Asta inseamna ca orice functie din fisier are acces sa citeasca si sa modifice schedulerul, lucru care nu imi convine, avand in vedere ca el trebuie accesat doar de functia loop, pentru metoda **check\_events().**

Drawback-ul ca la prima rulare a loop-ului timpul e putin mai mare si mai executa un if in plus la urmatoarele rulari, sau faptul ca e nu e la fel de readable nu mi se par drawbackuri suficiente cat sa pun **static scheduler,** in afara functiei.

Am facut initializarile in stilul acesta peste tot, si am lasat variabile **statice la nivel de fisier** doar unde mi s-a parut necesar.

**2.11 REQ010: Security start**

The application should start only if a code is introduced from a keypad or a IR remote control.

Folosim un keypad 4x3. Daca codul cerut la pornirea aplicatiei este corect trecem la aplicatiei altfel se afiseaza cod invalid si se cere reintroducerea.

Detaliam **scanarea tastaturii** fiindca e mai interesanta, functia e adaptata de la lab-ul domnului Lemeni.

char

*kbscan\_no\_debounce*(void)

{

*PORTD* = 0;

*u8* temp;

*//u8 cols;*

*u8* lina = 0; *//linia activa*

*u8* cola = 0; *//coloana activa*

    char tabela\_trans[] =

    {

        "123A456B789C\*0#D"

    };

    for(*DDRD* = (1 << 3) | (1 << 7); *DDRD* != (1 << 7); *DDRD* = (*DDRD* << 1) | (1 << 7), lina++)

    {

        temp = ~*PIND* & 0b111;

        if (temp)

        {

            cola = 0;

            for (; temp != 1; temp >>= 1, cola++)

            ;

            return tabela\_trans[3 \* lina + cola];

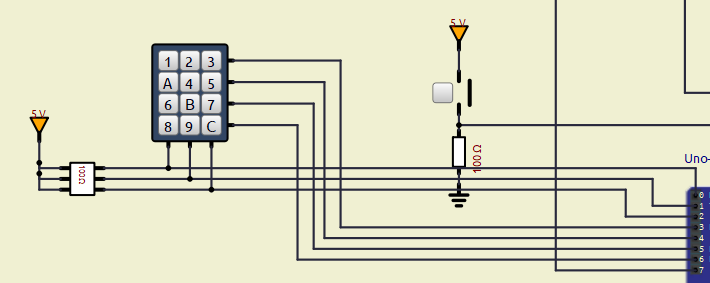
        }

    }

    return *NOKEY*;

}

Montajul:



Liniile sunt conectate pe pinii 3,4,5,6. Pe rand conectam unul dintre acesti pini ca iesire, ceilalti fiind ca intrari, deci impedanta inalta la intrarea in MCU. **PORTD = 0**, deci pinul pe care il conectam ca iesire trimite LOW pe linia corespunzatoare. Daca tasta este apasata atunci **pe coloana corespunzatoare tastei apasata va fi LOW**. Pe celelalte coloane va fi **HIGH**. Coloanele sunt conectate pe 0,1,2. For-ul:

for(*DDRD* = (1 << 3) | (1 << 7); *DDRD* != (1 << 7); *DDRD* = (*DDRD* << 1) | (1 << 7), lina++)

seteaza un singur pin de la 3 la 6 ca iesire (si tot timpul pe pinul 7 care e conectat la actuator, deci trebuie sa fie iesire, btw nu afecteaza actuatorul fiindca PORTD = 0), si totodata incrementeaza linia curenta. Apoi citim valoarea de pe coloane si o negam ca tasta apasata sa fie recunoscuta cu “1”: **~PIND & 0b111**. Apoi **cola** este pozitia celui mai din dreapta (putin semnificativ) “1”. Si din lina si cola deducem caracterul din Look Up Table. (tabela\_trans[3 \* lina + cola]).

Daca nu am gasit niciun “1” pe nicio linie returnam **NOKEY**.

Functia fiindca returneaza un singur char **nu poate detecta daca mai multe taste au fost apasate**, lucru usor de schimbat in viitor.

**Debouncing** (tot asa mi s-a parut mai interesant de prezentat):

char

*kbscan*(void)

{

    static *u32* ts\_prev = 0;

    static char ch\_prev = *NOKEY*;

    static const *i16* DEBOUNCE\_DELAY = 30;

    char result = *NOKEY*;

    if (*millis\_*() - ts\_prev >= DEBOUNCE\_DELAY)

    {

        char ch\_now = *kbscan\_no\_debounce*();

        if (ch\_prev == *NOKEY* && ch\_now != *NOKEY*)

        {

            result = ch\_now;

        }

        ts\_prev = *millis\_*();

        ch\_prev = ch\_now;

    }

    return result;

}

Bouncing-ul a butoanelor tasaturii este o perioada scurta in care de la o tranzitie (PRESSED -> NOT\_PRESSED sau NOT\_PRESSED -> PRESSED), butonul nu face tranzitia clean, ci oscilieaza in continuu intre LOW si HIGH (NOT\_PRESSED si PREESSED), pana ce eventual se stabilizeaza.

Functia face citiri aprox. din 30 in 30 de ms. (la cel putin 30 si depinde cat de des e apelata). Daca apelezi la mai putin de 30ms de la ultimul apel se returneaza NOKEY.

Cum ajuta asta cu bouncing-ul? Pai problema cu bouncing-ul e ca putem detecta o apasare de mai multe ori. Noi vedem o apasare ca la ultimul apel nimic sa nu fie apasat (sa returneze NOKEY) si la apelul curent o tasta sa fie apasata. Astfel nu detectam acelasi press de mai multe ori (sa zicem ca tasta ‘A’ e apasata pentru 100ms si in timpul asta facem 10 citiri, dar returnam o singura data caracterul ‘A’). Bouncing-ul ne creeaza tranzitii false de tipul NOT\_PRESSED ->PRESSED si atunci am recunoaste de mai multe ori caracterul apasat.

Solutia e sa facem citirile la interval mai mare decat perioada de bouncing dar mai mici decat perioada minima de apasare (in mod normal un om nu poate apasa mai putin de 50ms si 2 apasari consecutive trebuie sa fie la distanta de cel putin 50ms).

Astfel in 2 citiri consecutive doar una poate fi in perioada de bouncing (cealalta trebuie sa fie stabila) si inseamna ca va lua tranzitia corect, **eventual cu un delay de 30ms, daca de exemplu butonul a fost la citirea anterioara NOT\_PRESSED, intre timp s-a apasat, dar noi am facut citirea in perioada de bouncing si s-a nimerit LOW(NOT\_PRESSED), vom putea detecta apasarea doar la urmatoarea citire peste 30ms, cand cu siguranta s-a stabilizat.**

Perioada de bouncing e in medie tot in intervalul acesta de 20-30ms, dar pe placa la laboratorul de la micro (la dl Lemeni) DEBOUNCE\_DELAY de 30ms a mers ok.

**2.2 REQ201: Central Door Locking Modes**

The doors shall have the following possible states:

- Locked

- Double-locked

- Unlocked

Sa facem o comparatie cum gestionam starile in varianta initiala, la prima factorizare si apoi cu state pattern:

typedef enum

{

    UNLOCKED =  0,

    LOCKED,

    DOUBLE\_LOCKED

} *door\_state*;

Starea este doar un enum si tratam actiunile (motoarele, ledurlie), timing-ul pentru butoane, intr-un switch dupa starea curenta, in care este imbricat un switch dupa starea butonului. Punem in prezentare doar ce se intampla la in cazul in care suntem in starea UNLOCKED:

  switch(state)

    {

        case UNLOCKED:

        {

*blink\_led*(UNLOCKED);

            static *u32* ts\_prev = 0;

*door\_in\_unlocked*(last\_state);

            *window\_in\_unlocked*(last\_state, door\_btn);

            if (last\_state != state)

            {

                ts\_prev = *millis*();

            }

            switch(door\_btn)

            {

                case NOT\_PRESSED:

                {

                    if (*test\_press*(*DOOR\_BTN*))

                    {

                        door\_btn = PRESSED;

                        ts\_prev = *millis*();

                    }

                }

                break;

                case PRESSED:

                {

                    if (*test\_release*(*DOOR\_BTN*))

                    {

                        door\_btn = NOT\_PRESSED;

                        if (*millis*() - ts\_prev >= 800)

                        {

                            last\_state = state;

                            state = LOCKED;

                        }

                    }

                }

                break;

            }

        }…

Codul face asa: a**prinde led-urile**, **pune un time-stamp** cand a intrat in starea curenta, **comanda actuatorii** cu 2 functii: ce face door in modul unlocked (door\_in\_unlocked()), respectiv window)\_in\_unlocked(), ambele primesc ca parametru last\_mode, care este nu ultimul mod (State) in care a fost usa inainte de modul curent, ci modul de la ultima iteratie.

**Deci functia de comanda a actuatorilor stie de state-uri, si nu poate fi folosita independent. La fel functia de blink depinde de mod.**

**Codul e intertwined: in acelasi loc: testez starea butonului si o schimb daca e necesar, citest pe last\_state si il setez pe update, iau time-stampuri, aprind leduri, lucrez cu actuatoare, vad timingul fata de apasarea butonului si vad daca trebuie sa trec in alta stare.**

**Nu respect DRY, duplic codul: la fiecare state din switch(mode) : {UNLOCKED, LOCKED, DOUBLE\_LOCKED} fac aceleasi lucruri cu last\_state si state, fac acelasi switch imbricat dupa buton etc.**

Cum imbunatateste prima refactorizare?

door\_state mode = *current\_state*();

*print\_state*(mode, door\_btn.state);

*update\_btn\_state*(&door\_btn);

    switch(mode)

    {

        case UNLOCKED: { *unlocked\_loop*(&door\_btn, &door\_lock, &window, mode);} break;

        case LOCKED: { *locked\_loop*(&door\_btn, &door\_lock);} break;

        case DOUBLE\_LOCKED: { *double\_locked\_loop*(&door\_btn, &door\_lock, &window);} break;

    }

In primul rand update-ul starii butonului se face o singura data, nu mai trebuie sa fac switch dupa starea butonului si sa testez press/release ca sa schimb starea **in fiecare mod. (cod duplicat).**

In al doilea rand am inglobat total ce se intampla in interiorul fiecarui mod in functii care primesc referinte la “**obiectele share-uite” adica butonul si actuatoarele**, in rest in aceasta versiune isi pastreaza starea static.

Se observa pe prima linie de cod ca starea curenta se obtine prin apelul **curent\_state(), fara niciun parametru**. Ceea ce indica ca starea este retinuta global cumva si ca este una singura per program.

O sa explic la varianta finala (a 2-a factorizare) de ce am ales ca starea programului sa fie **singleton.**

Momentan, sa vedem cum arata una dintre functiile care fac actiunile corespunzatoare unui mod, de ex: **locked\_loop()** (am pus numele loop, nu fiindca sunt un loop, ci fiindca se executa in loop, nu prea e sugestiv)

void

*locked\_loop*(btn\_info\* door\_btn, actuator\* door\_lock)

{

    static timer timer\_;

    static blinking\_leds leds;

    static bool\_ init = 0;

    static bool\_ start\_new\_rotation = FALSE;

    if (!init)

    {

        init = TRUE;

        leds = *init\_leds*(LOCKED);

*reset*(&timer\_);

    }

    if (*entered\_new\_state*())

    {

*reset\_leds*(&leds);

*reset*(&timer\_);

       start\_new\_rotation = TRUE;

    }

    else

    {

        start\_new\_rotation = FALSE;

    }

*rotate\_cw*(door\_lock, 1000, start\_new\_rotation);

*blink*(&leds);

    if(*pressed*(door\_btn))

    {

*reset*(&timer\_);

    }

    if (*released*(door\_btn))

    {

        Serial.*println*("Am dat release la buton in modul locked.");

        if (*have\_passed*(timer\_, 100) && !*have\_passed*(timer\_, 800))

        {

*update\_state*(DOUBLE\_LOCKED);

        }

    }

    if(door\_btn->state == PRESSED && *have\_passed*(timer\_, 800))

    {

        Serial.*println*("Trec in unlocked.");

*update\_state*(UNLOCKED);

    }

}

Lasam momentan de o parte actuatoarele, vedem ca codul se imparte in 3 stagii:

1.ce execut cand am intrat in stare

2.ce execut cat timp sunt in stare (de fiecare data cand intru)

3.intreb daca s-a indeplnit conditia sa ma mut in alta stare si daca da: ma mut

Concret:

1.resetez ledurile si timerul (detaliez imediat cum functioneaza timerul)

2.apelez blink: patternul la led-uri, vad daca butonul e apasat si resetez timerul

3. daca butonul s-a dat release la buton si au trecut intre 100 si 800 ms schimb starea in DOUBLE\_LOCKED, daca butonul e apasat si au trecut 800ms atunci intram in UNLOCKED

**Obs:**

if(door\_btn->state == PRESSED && *have\_passed*(timer\_, 800))

    {

        Serial.*println*("Trec in unlocked.");

*update\_state*(UNLOCKED);

    }

La o prima impresie codul acesta ar putea trece in UNLOCKED daca butonul nu ar fi fost apasat continuu, ci doar la un moment de timp la cel putin 800ms de la resetarea timerului. Acest lucru nu se poate intampla findca in contextul in **care door\_btn->state == PRESSED, trebuie ca functia pressed(door\_btn) sa detecteze press-ul (trecerea din NOT\_PRESSED in \_PRESSED) si sa reseteze timerul.** Deci in momentul in cate imi intra pe update\_state(UNLOCKED), au trecut cel putin 800 de ms de la ultimul release.

**Obs: Eventual bug:** Fie urmatorul scenariu. Trecem dintr-un mod: sa zicem UNLOCKED in alt mod in momentul in care led-urile sunt pe ON. Daca modul in care trecem nu reseteaza

ledurile, atunci led-ul va ramane pe ON. In prezent toate modurile reseteaza ledurile cand s-a intrat in modul respectiv, dar pe viitor daca adaugam moduri se poate sa introducem un mod care sa nu aiba treaba cu led-urile.

La fel si cu actiunile pornite pe actuator.

**Solutia este sa dam cleanup() la actiunile pe care le-am inceput in momentul in care trecem in alta stare.**

Sa vedem cum arata singletonul de state si functiile care lucreaza pe stari si apoi trecem la avantajele factorizarii cu **state pattern.**

#include "headers/door\_states.h"

static bool\_ entered\_new\_mode = FALSE;

static door\_state mode = UNLOCKED;

bool\_

*entered\_new\_state*(void)

{

    if (entered\_new\_mode)

    {

        entered\_new\_mode =  FALSE;

        return TRUE;

    }

    return FALSE;

}

void

*update\_state*(door\_state *new\_mode*)

{

    if (*new\_mode* != mode)

    {

        entered\_new\_mode = TRUE;

        mode = *new\_mode*;

    }

}

door\_state

*current\_state*(void)

{

    return mode;

}

Deci avem o variabila “gloabala” (accesibila la nivel de fisier) in care retinem starea curenta.

Din alte fisiere starea este bine incapsulata, adica este accesibila modificarea doar prin metoda de **update\_state()**.

Cu observatiile anterioare: impartirea actiunii executate intr-un mod in: executie la intrarea in stare, executie permanenta cat timp sunt in stare, verificare daca trebuie sa trec in starea urmatoare (cu timing si detect de press/release) si cleanup().

Si cu observatia ca varabilele statice din interiorul functiilor: unlocked\_loop() fac grea testarea (mocking-ul) si nu incapsuleaza bine datele, vedem cum **state pattern** **imbunatateste design-ul codului**.

Pentru a introduce scurt state pattern, dau un paragraf din **Patterns in C** de **Adam Tornhill:**

The main idea captured in the STATE pattern is to represent each state as an object of its own. A state transition simply means changing the reference in the context (*DigitalStopWatch*) from one of the concrete states to the other.

**Implementation Mechanism**

Which mechanism may be suitable for expressing this, clearly object oriented idea, in C? Returning to our example, we see that we basically have to switch functions upon each state transition.

Luckily, the C language supplies one powerful feature, pointers to functions, that serves our needs perfectly by letting us change the behavior of an object at run-time. Using this mechanism, the interface of the states would look as:

Listing 1: The state interface in WatchState.h

1 */\* An incomplete type for the state representation itself. \*/*

2 **typedef struct** WatchState\* WatchStatePtr;

3

4 */\* Simplify the code by using typedefs for*

5 *the function pointers. \*/*

6 **typedef** void (\*EventStartFunc)(WatchStatePtr);

7 **typedef** void (\*EventStopFunc) (WatchStatePtr);

8

9 **struct** WatchState

10 {

11 EventStartFunc start;

12 EventStopFunc stop;

13 };

**La refactorizarea finala: switch-ul dupa mod e inlocuit de handle\_state() fara parametri (state global) dupa cum urmeaza:**

**Main\_task.cpp:**

void

*main\_task*(void)

{

    static *btn\_info* door\_btn;

    static *actuator* door\_lock, window;

    static *bool\_* init = *FALSE*;

    if (!init)

    {

        init = *TRUE*;

        door\_btn *=* *btn\_init*(*DOOR\_BTN*);

        door\_lock *=* *actuator\_create*(*DOOR\_LOCK\_MINUS*, *DOOR\_LOCK\_PLUS*);

        window *=* *actuator\_create*(*WINDOW\_MINUS*, *WINDOW\_PLUS*);

*init\_states*(&door\_btn, &door\_lock, &window);

*update\_state*(UNLOCKED); *// starea initiala*

    }

*lcd\_print\_state*(*current\_state*(), door\_btn.state);

*update\_btn\_state*(&door\_btn);

*handle\_state*();

}

Initializez butonul si cei 2 actuatori si ii dau ca parametrii pentru init\_states() care va

initializa toate cele 3 stari: LOCKED, UNLOCKED, DOUBLE\_LOCKED, dandu-le un handle catre buton si actuatori.

Apoi by default trecem in starea UNLOCKED. Eventual, dar e mai complicat de testat fiindca nu se permite in simulare, putem sa scriem in ROM (Flash) cu **progmem**, starea curenta si la boot sa o ia de acolo. (ca tot timpul sa porneasca de unde a ramas).

Bun. Acum sa vedem ce face handle\_state():

void

*handle\_state*(void)

{

    if (*entered\_new\_state*())

    {

        state\_->*on\_enter\_state*(state\_);

    }

    state\_->*while\_in\_state*(state\_);

*door\_mode* new\_mode;

    if (state\_->*transition\_cond*(state\_, &new\_mode))

    {

        state\_->*on\_exit\_state*(state\_);

*update\_state*(new\_mode);

    }

}

Face toti pasii descrisi mai sus:

1.se uita daca a trecut in stare noua (starea din care se apeleaza e stare noua) si executa functia on\_enter\_state() care ia ca parametru **state\*** (**state\_** e de tipul **state\***). Functia on\_enter\_state() este de fapt pointer la functie, adica in momentul in care declaram o stare (variabila de tipul state) ii putem da ce actiuni sa ruleze.

Actiunile din cerinta au nevoie si de variabile din exterior

Obligatoriu vorbeste de actuatori. Sa spargi functia rotate in start\_rotate() care sa bage o comanda in actuator si apoi functia rotate cu bool pe FALSE sa fie de fapt monitor care sa vada daca comanda s-a sfarsit, a trecut timpul si sa dea comanda de stop la actuator.