CDL – Key points

Cuprins:

1. Leduri/Actuatori: Executarea functiilor “pe bucati”. Cum ne-ar ajuta un OS:

Luam functia de care face **patternul ledurilor**:

Daca ar fi fost singurul lucru care rula pe sistem, implementarea ar fi putut arata asa:

blink():

```
for (i = 0 ; i < nr_blinks; i++)
{
        leds_on();
        delay(300);
        leds_off();
        delay(700)
}
```

In sistem totusi loop-ul nostru, are si alte output-uri: **actuatorii** si trebuie sa fie tot timpul **responsive la buton** ca sa schimbe starea si implicit ledurile si actuatorii.

Solutia este ca in loc sa apelam **blink()** o singura data ca sa fac tot patternul,

o apelam in **bucla** si la fiecare pas face o parte foarte **mica din pattern**.

Ne uitam pe implementarea din varianta **nefactorizata:**

```
void

blink_led(door_state mode)
{
    static door_state last_mode = UNLOCKED;
    static u8 count = 0, cycles = 0, nr_cycles = 0;
    static u8 nr_blinks[] = {2, 1, 3};
    if (last_mode != mode)
    {
        cycles = 0;
        count = 0;
        nr_cycles = nr_blinks[mode];
        last_mode = mode;
    }
    //digitalWrite(10, HIGH);
    static u32 ts_prev = 0;
    if (cycles < nr_cycles)
```

```
    {
        if (millis() - ts_prev >= 100)
        {
            if (count == 0)
            {
                digitalWrite(LED0, HIGH);
                digitalWrite(LED1, HIGH);
            }
            if (count  ==  3)
            {
                digitalWrite(LED0, LOW);
                digitalWrite(LED1, LOW);
            }

            if (count++ == 10 - 1)
            {
                count = 0;
                cycles++;
            }
            ts_prev = millis();
        }
    }
}
```

Se observa ca luam o decize o data la **100 ms,**

intreband la fiecare intrare in functie daca au trecut 100ms de **ultima data cand am facut o actiune** (**ts_prev = time_stamp_previous**)

la fiecare trecere de 100ms, updatam variabilele care ne spun unde ne aflam in executia patternului (ex: cycles = 1, count = 4, am facut un blink complet sunt 400ms intr-al doilea)

In functie de ele luam actiuni:

cand incepem un un blink (count == 0) facem **leds_on()**

cand count == 3 (300ms intr-un blink) facem **leds_off()**

cand cycles >= nr_blinks: am terminat de executat, apelurile urmatoare nu fac nimic pana:

reset_state cand se schimba modul (last_mode != mode)

Cat de repede trebuie sa apelam functia in loop?

Pai daca am apela o data la 100ms si ultimul apel a fost sa zicem la **299ms** intr-un blink :exact inainte sa trebuiasca sa facem **leds_off()**. Urmatoarea data cand intram si facem **leds_off()** au trecut cel putin 400ms, deci noul **FU = 40%, desi trebuia sa avem FU = 30%**

Asadar functiile “pe bucati” trebuie apelate foarte des si in consecinta,

timpul total pe care il petrecem in **WHILE(1), daca avem “functii pe bucati” trebuie sa fie foarte mic.**

In concluzie, trebuie ca fiecare functie apelata de **while(1)** sa fie suficient de “sparta in bucati”, adica **sa faca suficient de putine lucruri intr-o singura iteratie ca sa nu consume timp.**

Functiile pe bucati trebuie sa retina starea executiei: in ce stadiu al executiei intregului sunt.

De ce nu e bine ca starea sa fie statica ca in exemplul nefactorizat?

Fiindca variabilele statice sunt unice per functie, indiferent de contextul in care apelam.

Sa presupunem ca in **loc de 2 leduri avem 4**. Si vrem sa luminam patternul cu primele 2 si la scurt timp cu celelalte 2.

Variabilele statice care imi spun statiul executiei **sunt unele singure si se vor amesteca**, setate prima oara de o pereche de leduri, citita eronat de a doua pereche crezand ca sunt variabilele ei etc.

Varianta factorizata: pune starea executiei intr-un struct pe care il paseaza mereu functiei la apel.

```
typedef struct
{
    u8 cycles, count, nr_blinks;
    timer led_timer;
    u8 pin0, pin1;
} blinking_leds;

void
blink(blinking_leds* leds)
{
    if (leds->cycles >= leds->nr_blinks)
    {
```

```
        return;
    }
    if (have_passed(leds->led_timer, 100))
    {
        reset_timer(&(leds->led_timer));
        if (leds->count == 0)
        {
            digitalWrite_(leds->pin0, HIGH);
            digitalWrite_(leds->pin1, HIGH);
        }
        else if (leds->count  ==  3)
        {
            digitalWrite_(leds->pin0, LOW);
            digitalWrite_(leds->pin1, LOW);
        }
        leds->count++;
        if (leds->count == 10)
        {
            leds->count = 0;
            leds->cycles++;
        }
    }
}
```

Asta rezolva problema, starii unice.

Potential bug: ce se intampla daca nu mai apelam **blink()** in timp in timp ce led-urile **sunt in mijlocul patternului**. (sa zicem trecem intr-un alt mod)

Pai daca **erau aprinse**, **raman aprinse** si va deveni responsabilitatea noului mod sa le stinga la inceput.

Dar daca modul nu are treaba cu led-urile? **Nu trebuie sa fie responsabilitatea unui alt mod sa curate dupa noi.**

Solutia este sa avem o functie de **cleanup()** pe care o apelam cand luam decizia ca **ne oprim din a apela functia** “pe bucati”. (in cazul nostru **reset_leds()**)

Deci functiile pe bucati au nevoie de

init(): ca sa-si reseteze starea cu stadul executiei la inceputul executiei unui task intreg

cleanup(): ca sa isi anuleze efectele in momentul in care a terminat sau a fost intrerupta

Exemplificam pe actuatoare (din src/devices/actuators.cpp):

Ne uitam ce se intampla in prima factorizare unde nu s-a respectat impartirea pe **init(), functia_propriu_zisa(), cleanup():**

Design-ul gresit al functiei rotate(actuator**, time, bool start_new_conversion):**

Pai sa vedem cum e apelata in cod:

```
void
unlocked_on_enter(state* me)
{
    actuator* door_lock = me->context.door_lock;
    actuator* window = me->context.window;
    blinking_leds* leds = &(me->context.leds);
    timer* timer_ = &(me->context.timer_);
    reset_leds(leds);
    reset_timer(timer_);
    u32 time = (last_state() == LOCKED) ? 1000 : 2000;
    rotate_counter_cw(door_lock, time, TRUE);
    rotate_cw(window, 3000, TRUE);
}
```

La intratarea in stare e apelata cu parametrul **start_new_rotation = TRUE,** cu semnificatia sa inceapa o rotatie.

Apoi se apeleaza continuu cat timp ramanem in stare cu parametrul **start_new_rotation = FALSE** aici:

```
void
while_in_unlocked(state* me)
{
    actuator* door_lock = me->context.door_lock;
    actuator* window = me->context.window;
    btn_info* door_btn = me->context.door_btn;
    blinking_leds* leds = &(me->context.leds);
    timer* timer_ = &(me->context.timer_);
    blink(leds);
    u32 time = (last_state() == LOCKED) ? 1000 : 2000;
    rotate_counter_cw(door_lock, time, FALSE);
    if (door_btn->state == PRESSED)
    {
        rotate_cw(window, 3000, FALSE);
    }
    else
    {
        stop_actuator(window);
    }
    if(pressed(door_btn))
    {
        reset_timer(timer_);
    }
}
```

Cele 2 apeluri au roluri diferite: primul apel cand am intrat in stare porneste o rotatie, urmatoarele apeluri vad daca rotatia s-a terminat si opresc actuatorul.

Comportament defectuos fiindca:

1. functia nu are o interfata intuitiva(**cine se uita la cod si vede bool_ start_new_conversion zice ce e asta?** si probabil nici daca citeste codul nu se lamureste**)**
2. nu are o singura responsabilitate, functia initiaza o rotire si monitorizeaza oprirea rotirii
3. Cel mai important permite urmatorul bug. am apelat rotate_cw(door_lock, **TIME = 1000, START_NEW_CONVERSION = TRUE)** si am pornit conversia. Mai tarziu inainte sa treaca cele 1000ms apelez rotate_cw(door_lock, **TIME = 500**, **START_NEW_CONVERSION = FALSE**). Desi initial am vrut sa pornesc o rotatie pentru 1000 ms, mai tarziu din greseala am apelat aceeasi functie cu 500ms si implicit rotatia va dura doar 500 de ms.
4. Alta problema este ca daca ies din stare fara sa treaca timpul de la **rotate_cw(),** actuatoarele raman pornite, solutia pentru asta este sa facem cleanup.

Pentru a rezolva aceasta problema spargem functia in 2 functii cu responsabilitate unica:

rotate() si **monitor().** Rotate incepe rotatia si seteaza o comanda de rotatie in struct-ul actuatorului si pornerste motorul. Monitor se uita pe comanda din monitor si daca a trecut timpul alocat comenzii opreste motorul(o comanda de rotire are un timp pentru care sa se roteasca)

```
typedef struct
{
    u32 exec_time;
    bool_ cw_0_ccw_1;
    bool_ active;
}  actuator_command;

typedef struct
{
    u8 minus_pin, plus_pin;
    timer timer_;
    actuator_command cmd;
}  actuator;
```

```
void
rotate(actuator* act, bool_ cw_0_ccw_1, u32 time)
{
    reset_timer(&(act->timer_));
    if (cw_0_ccw_1)
    {
        digitalWrite_(act->plus_pin, !cw_0_ccw_1); // obligatoriu incepem cu low ca sa nu
scriem high pe ambii pini
        digitalWrite_(act->minus_pin, cw_0_ccw_1);
```

```
    }
    else
    {
        digitalWrite_(act->minus_pin, cw_0_ccw_1);
        digitalWrite_(act->plus_pin, !cw_0_ccw_1);
    }
    actuator_command cmd;
    cmd.exec_time = time;
    cmd.cw_0_ccw_1 = cw_0_ccw_1;
    cmd.active = TRUE;
    act->cmd = cmd;
}

void
monitor_actuator(actuator* act)
{
    if (act->cmd.active && have_passed(act->timer_, act->cmd.exec_time))
    {
        stop_actuator(act);
    }
}
void
rotate_cw(actuator* act, u32 time)
{
    rotate(act, CW, time);
}

void
rotate_counter_cw(actuator* act, u32 time)
{
    rotate(act, CCW, time);
}
```

Cum ne-ar ajuta un OS cu functiile “pe bucati”?

Functiile “pe bucati“ se ruleaza in bucla, isi **retin progresul din executia “intregului”** si in functie de **variabilele de progres decid care e urmatorul pas de executat**.

Programarea in stilul asta e greoaie si neintuitiva.

Sa ne gandim cum ar fi sa scriem “pe bucati” ceva mai complicat cum ar fi o sortare, sa zicem **merge_sort().**

Merge sort sorteaza recursiv ambele jumatati ale vectorului si le interclaseaza.

Cum punem varabilele de progres ca in functie de ele sa vedem care ar fi intervalul curent pe care il sortam (din stiva de recursivitate).

Cum vedem daca daca **urmatorul pas “mic”** e un pas dn **splitul intervalului**, un pas din **sortarea unui interval,** un pas din **interclasare.**

E clar ca e **mult mai greu de scris** decat **varianta secventiala.**

Ideea e ca orice **PC** trebuie sa **fie tot timpul responsive la inputuri** (de la kbd, de la network etc.) si are tot timpul mai multe programe care **genereaza mai multe output-uri** (pe ecran sa zicem), deci se loveste cu **acelasi probleme ca aplicatia noastra**.

Pe PC putem scrie codul **secvential, nu "pe bucati"** si totusi efectul e cel dorit.

Solutia de pe PC e ca OS-ul imparte tot ce se intampla pe procesor in **taskuri** (si apoi taskurle/procesele in thread-uri etc.).

La o perioada foarte scurta de timp, sa zicem **1ms (**probabil si mai mica nu stiu**),**

OS-ul executa o **intrerupere** pe timer care se uita la ce procese incearca sa ruleze "simultan" si in functie de **prioritati** si alte criterii, ii da procesorul pana la urmatoarea intrerupere unuia dintre dintre **procese**.

Astfel codul care are nevoie sa fie **responsive** se poate executa suficient de rapid.

Iar faptul ca avem mai multe output-uri se poate rezolva prin executarea lor intr-un thread separat (al aceluiasi proces).

Cu un OS, ruland sistemul pe ma`i **multe thread-uri**:

am fi putut scrie functia blink secvential nu "pe bucati" asa

<u>Thread leds:</u> blink():

```
for (i = 0 ; i < blinks; i++)
{
        leds_on();
        delay_neblocant(300);
        leds_off();
        delay_neblocant(700);

}
```

delay_neblocant(time_) inseamna: daca n-au trecut **time_ ms** de la apel **da control altui thread (sau scote-ma din lista de threaduri care vor sa se execute pana cand au trecut time_ ms)**

Apoi, putem intrerupe functia **blink()** prin: **stop(Thread_leds),** si ca de obicei trebuie sa dam **cleanup().**

2.FSM. State Pattern. Simulare OOP in C

Incepem cu managementul starilor in **varianta nefactorizata.**

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**:

```
typedef enum
{
    UNLOCKED =  0,
    LOCKED,
    DOUBLE_LOCKED
} door_state;
```

```
    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;
```

Codul responsabil de managementul starilor 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 starea curenta si anterioara.

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.

Ne uitam la implementarea functiiilor care imi dau starea in **src/fsm/door_states.cpp.**

O sa explic la varianta finala (a 2-a factorizare) de ce am ales ca starea programului sa fie una singura per program(**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” (statica accesibila la nivel de fisier) in care retinem starea curenta.

Din alte fisiere starea este bine **incapsulata**, adica se poate modifica doar prin **update_state()** si cum noi avem **un singur FSM**, este o solutie buna.

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**.

State Pattern:

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**.

De exemplu la intrarea in starea **UNLOCKED**, avem nevoie sa stim **starea anterioara pentru a stii daca rotim 2s sau 1s.** In permanenta trebuie sa stim starea butonului ca sa declansam tranzitia catre o alta stare etc.

Pentru a nu retine informatiile acestea ca statice in interiorul functiilor pentru fiecare mod le punem intr-o structura numita **state_context.**

```
typedef struct
{
    btn_info* door_btn;
    actuator* door_lock, *window;
    blinking_leds leds;
    timer timer_;
} state_context;
```

Simulare OOP in C:

Dupa cum s-a discutat despre state pattern fragmentul din **Patterns in C,** incercam sa emulam mostenirea din OOP in C.

Intr-un limbaj OOP, am putea crea o interfata / clasa abstracta **State** care sa aiba metode vitualae pentru ce sa executa la intrarea in stare ,cat timp e in stare etc. O stare concereta, cum ar fi **UNLOCKED** ar mosteni **State,** si ar implementa toate aceste metode.

In C putem sa emulam acest mecanism cu pointer la functii (pana la urma **tabelul de functii virtuale este doar un set de ponteri la functii)**

Definim structul **state** (care ar corespunde **interfetei State in OOP**):

```
typedef struct state
{
    door_mode mode;
    state_context context;
    void (*on_enter_state)(state* me);
    void (*while_in_state)(state* me);
    bool_ (*transition_cond)(state* me, OUT_PARAM door_mode* new_mode);
    void (*on_exit_state)(state* me);
} state;
```

Are pointeri la functie pentru toate situatiile discutate.

Se observa parametrul numit **“me”**. A metoda se apleaza cu **me->on_enter_state(me),**

care e echivalent intr-un limbaj **OOP, sa zicem C++** cu **this->on_enter_state();**

In particular metoda **transition_cond(),** returneaza **TRUE** daca trebuie sa facem tranzitie si in **new_mode** (parametru de iesire) vedem in ce stare trebuie sa mergem si apelam **update_state()** de starea respectiva.

Sa vedem unde si cum se instantiaza “clasele concrete” adica starile concrete (un singleton pentru fiecare stare concreta UNLOCKED/LOCKED/DOUBLE_LOCKED).

Sa ne uitam la metoda de initializare pentru UNLOCKED:

```
state
create_unlocked_state(btn_info* door_btn, actuator* door_lock, actuator*
window)
{
    state unlocked_state;
    unlocked_state.mode = UNLOCKED;
    unlocked_state.on_enter_state = unlocked_on_enter;
    unlocked_state.while_in_state = while_in_unlocked;
    unlocked_state.transition_cond = transition_cond_unlocked;
    unlocked_state.on_exit_state = unlocked_on_exit;
    state_context context;
    context.door_btn = door_btn;
    context.door_lock = door_lock;
    context.window = window;
```

```
    context.leds = init_leds(UNLOCKED);
    reset_leds(&context.leds);
    reset_timer(&context.timer_);
    unlocked_state.context = context;
    return unlocked_state;
}
```

Acesta este “constructorul” “clasei” (in emularea noastra de OOP in C), facem ca pointeri la functie sa pointeze la metode concrete si ii initializam obiectele din context (buton, actuatoare) sa pointeze la **obiectel share-uite de toate modurile care sunt initializate in main_task().**

Vedem concret cum efectuam actiunile folosind **contextul starii**, sa zicem in starea **UNLOCKED:**

```
void
unlocked_on_enter(state* me)
{
    actuator* door_lock = me->context.door_lock;
    actuator* window = me->context.window;
    blinking_leds* leds = &(me->context.leds);
    timer* timer_ = &(me->context.timer_);
    reset_leds(leds);
    reset_timer(timer_);
    u32 time = (last_state() == DOUBLE_LOCKED) ? 2000 : 1000;
    rotate_counter_cw(door_lock, time);
    rotate_cw(window, 3000);
}

void
while_in_unlocked(state* me)
{
    actuator* door_lock = me->context.door_lock;
    actuator* window = me->context.window;
    btn_info* door_btn = me->context.door_btn;
    blinking_leds* leds = &(me->context.leds);
    timer* timer_ = &(me->context.timer_);
    blink(leds);
    monitor_actuator(door_lock);
    if (door_btn->state == PRESSED)
    {
        monitor_actuator(window);
    }
    else
    {
        stop_actuator(window);
    }
```

```
    if(pressed(door_btn))
    {
        reset_timer(timer_);
    }
}

bool_
transition_cond_unlocked(state* me, OUT_PARAM door_mode* new_mode)
{
    btn_info* door_btn = me->context.door_btn;
    timer timer_ = me->context.timer_;

    if(released(door_btn) && have_passed(timer_, 800))
    {
        *new_mode = LOCKED;
        return TRUE;
    }
    return FALSE;
}

void
unlocked_on_exit(state* me)
{
    default_on_exit(me);
}
```

Obs: Pentru usurinta am facut o serie de metode default pe care pot sa le utlizez in momentul in care o stare are comportament default in acea situatie. (cum se sugereaza in cartea **Patterns in C**).

Sa vedem cum se executa acest cod pentru a indeplini requirement-urile. Sa zicem:

2.3 REQ202: Activation of Locked mode

When the user presses the lock switch once for at least 800ms and release it again, the doors shall be locked if the doors are currently unlocked.

Ne intereseaza ca in functia while_in_unlocked() apelata la fiecare iteratie cand sunt in stare timerul este resetat cand butonul este apasat (functia **pressed()**).

Apoi functia **transition_cond_unlocked()** verifica daca butonul a fost released() (lucru care se intampla o singura data cand s-a dat release, adica pentru **un singur release** functa released() returneaza **TRUE o singura data**) si verifca daca timerul ajuns la 800 de ms,

de cand a fost ultima oara resetat. Resetul se intampla **ori pe pressed()** ori **la intrarea in stare**, deci in momentul in care avem timerul pe 800 de ms si conditia de released(), cu siguranta butonul a fost apasat cel putin 800 de ms inainte. => **cerninta de trecere in modul locked este verificata**

Celelalte cerinte de trecere in moduri se verifica similar in cod, uitandu-ne la timing si la starea butonului. Se verifica doar functile de while_in_[un/ /double]locked(), respectiv, transition_cond_[un/ /double]locked().

3.HAL. Mocking. Testare leduri.

Hardware Abstraction Layer:

Ne uitam putin pe codul din **lcd.cpp**:

```
static LiquidCrystal lcd(RS, EN, D4, D5, D6, D7);

void
lcd_init()
{
    pinMode_(RS, OUTPUT);
    pinMode_(EN, OUTPUT);
    pinMode_(D4, OUTPUT);
    pinMode_(D5, OUTPUT);
    pinMode_(D6, OUTPUT);
    pinMode_(D7, OUTPUT);
    lcd.begin(16, 2);
    delay(10);
    lcd.clear();
    delay(10);
}

void
lcd_print(const char text[])
{
    lcd.print(text);
}

void
lcd_print_char(char ch)
{
    lcd.print(ch);
}

void
lcd_setCursor(u8 col, u8 lin)
{
    lcd.setCursor(col, lin);
}
```

In primul rand LCD-ul este **singleton** in program, dar mai important:

Este accesat cu un **wrapper** in jurul obiectului **LiquidCrystal** din < **LiquidCrystal.h>.**

De exemplu **lcd.print()** este **wrapped** de **print_lcd()**.

De ce? Ca sa nu depinda codul meu de librarii externe.

Observatie: Codul actual nu depinde deloc de nicio librarie externa, in afara de setup() si loop() nu depinde deloc nici de <Arduino.h>.

Btw, daca le inlocuiam cu **main()** si cu **while(1)** nu mergea simularea.

Toate dependentele sunt **indirecte**, de exemplu codul meu foloseste mills_() (varianta implementata de mine, nu cea din Arduino.h), **millis_()** depinde in spate de **<avr/io.h>** si de **<avr/interrupt.h>,** dar functionalitatea de baza nu depinde de ele.

Pot sa compilez programul principal separat intr-un fisier obiect **.o** si acel fisier sa stie doar ca are nevoie de o functie **millis_()** care returneaza un **u32.** Implementarea concreta a lui **millis**_ care foloseste in spate **<avr/interrupt.h>** poate fi compilata separat si cele 2 fisiere obiect pot sa fie legate in executabil **la linkare**.

De ce e util sa nu depind de lucruri concrete cum ar fi librarii sau hardware specific?

1.Ne gandim ce se intampla daca **folosim un alt tip de LCD**, sa zicem unul care foloseste **I2C.** Eu trebuie sa recompilez tot codul fiindca codul meu stie doar de **lcd(RS, EN, D4, D5, D6, D7),** si acum poate ca biblioteca mea care **imi face in spate protocolul cu led-ul** poate **nu se mai apeleaza cu lcd.print(),** poate se apleaza **cu lcd.write()**, sa zicem.

Acum trebuie sa inclocuiesc in cod peste tot **lcd.print()** cu **lcd.write()** si sa recompilez tot codul. Daca am un wrapper pot doar sa inlocuiesc pe **print_lcd()** sa apeleze in spate pe **lcd.write()** in loc de ce aplela inainte: **lcd.print()**.

2. Cum testam? Daca vreau sa fac testing nu pe placa, ci sa testez logica pe un **PC.**

De ce as testa nativ pe PC?

1. pot sa testez logica programului cu un **debugger mult mai avansat** (gen **Visual Studio**).

Motive luate din **Test Driven Development for Embedded C** (carte):

2. de multe ori **hardware-ul si software-ul se dezvolta simultan** si nu avem acces in timpul dezvoltarii de cod la hardware-ul pe care o sa ruleze
3. vreau sa dezvolt ceva care sa poate fi rulat pe **mai multe interfete hardware** fara sa schimb codul radical de la o interfata la alta

Bun. **Ce trebuie sa fac ca sa testez pe PC?**Pai am nevoie sa inlocuiesc tot ce **e hardware dependent** (Prin bibliotecile de AVR) si sa le inlocuiesc cu construct-uri care exista pe PC. De exemplu LCD e terminalul. Apasarea de buton e apasarea unei taste etc.

Daca am **dependente de hardware clare** in cod trebuie **sa le inlocuiesc peste tot** cu ceva abstractie care sa poata fi simulata pe PC. Deci trebuie **sa stau si sa caut in cod toate apelurile direct la hardware**, sa le inlocuiesc cu chestii abstracte. Dupa ce trece ceva

timp si vreau sa testez o alta functionalitate trebuie iar sa inlocuiesc toate apelurile directe la hardware din cod.

Imi este foarte greu sa fac **mocking** (sa inlocuiesc parti reale de cod cu **comportament simulat**) pentru testare daca depind **direct de HW.**

Solutia ca sa nu trec prin toate astea este sa fac un **Hardware Abstraction Layer.**

Pe scurt**, toate call-urile care lucreaza direct cu hardware sa fie inglobate in niste functii wrapper.**

Astfel logica prinicipala a codului poate fi **compilata ca un modul separat**, apelurile la functiile wrapper find **dispachuite la linkrae** fie catre **call-urile directe la HW**, fie catre **mocking-urile mele pentru testare**, fie la call-uri catre **o alta componenta HW fara sa mai modificam cu nimic functionalitatea principala.**

Testarea modulului de led-uri:

Ne uitam la **dependente**:

leds.cpp include **leds.h:**

```
#include "devices/leds.h"
```

leds.h include **utils.h, timer.h, door_states.h si registers.h:**

```
#include "utils/utils.h"
#include "utils/timer.h"
#include "fsm/door_states.h"
#include "utils/registers.h"
```

Implementarea timerului din **timer.cpp** e **platform independent.**

Din **utils.h** luam doar macrouri si typedef-uri care sunt **platform indepentent**.

Din **door_states.h** iau doar un enum.

Problema apare cu **registers.h** care ofera semnaturulile functiilor **millis_(), pinMode_(), digitalRead/Write(),** fiindca **registers.cpp** ofera implementari cu registre AVR, deci nu **poate fi executata pe PC**:

```
#include "utils/registers.h"
#include <avr/io.h> //pentru PORT,DDR etc.
#include <avr/interrupt.h>
```

Solutia este sa simulam comportamentul functiilor (**mocking)** din **registers.h** cu niste implementari care pot rula pe **PC**.

Pinii microcontrollerului pot sa fie **un vector in memorie**,

digitalRead_/Write_(): pot sa citeasca/scrie acolo.

pinMode_(): poate sa nu faca nimic in testele pentru leduri fiindca toti pinii vor fi by default de iesire.

millis_(): putem folosi o biblioteca de masurat timp in aplicatii pe **PC**

test/mocked_registers.cpp:

```
static u8 pins[NR_PINS] = {0};

void
digitalWrite_(u8 pin, u8 val)
{
    pins[pin] = val;
}

u8
digitalRead_(u8 pin)
{
    return pins[pin];
}

void
pinMode_(u8 pin, u8 mode)
{
    return; //toti pinii sunt de iesire nu ma mai complic
}

u32
millis_(void)
{
    //LUAT DE PE GEEKSFORGEEKS
    // Get the current time from the system clock
    auto now = chrono::system_clock::now();

    // Convert the current time to time since epoch
    auto duration = now.time_since_epoch();

    // Convert duration to milliseconds
    auto milliseconds
        = chrono::duration_cast<chrono::milliseconds>(
            duration)
            .count();
    return milliseconds;
}
```

Mai ramane sa confguram proiectul ca la build-ul **native** (pe PC), la linkare **registers.h** sa nu ia imlpementarile din **registers.cpp**, ci din **mocked_registers.cpp.**

Acest lucru depinde de build tool / IDE.

Cu **mockul** facut putem sa scriem testele pentru leduri:

Testam daca un led e aprins vazand daca pinul respectiv este HIGH.

Testam ledurile, sa zicem, pe modul **DOUBLE_LOCKED** unde trebuie sa avem **3** luminari de **1s** cu **FU = 30%.** (300 ms ON, apoi 700ms OFF)

Ca tool pentru testing folosim **Unity**, framework pentru testare in C/C++

Testele in Unity pot fi scrise si pentru **Arduino Uno**, nu doar nativ pe **PC**.

Nu folosim decat functionalitatile basic:

-setUp(): se ruleaza inainte de executia fiecare test, resetand starea ledurilor, a pinilor si timerul cu care verificam cat timp a trecut ca sa dam assert la momentul corect:

```
void
setUp(void)
{
    for (u8 pin = 0; pin < 28; pin++)
    {
        digitalWrite_(pin, LOW);
    }
    leds = init_leds(DOUBLE_LOCKED);
    reset_timer(&timer_);
}
```

-teardDown(): face cleanup dupa fiecare test, in cazul nostru noi facem cleanup in setUp(), asa ca nu facem nimic in **tearDown().**

-***TEST_ASSERT_TRUE():*** verifica daca **expresia e diferita de 0 (true**), daca ai mai multe assert-uri testul pica in momentul in care **unul din ele pica**. (am respectat totusi un singur assert)

-**R*UN_TEST():*** rulam un test care trebuie sa fie o functie care primeste si returneaza void

-**UNITY_BEGIN(), UNITY_END():** initializari / cleanup, intre ele scriem testele

Facem 2 tipuri de teste:

1.daca la momentul de timp **t** ledurile sunt aprinse sau nus

2.daca pe tot intervalul **[t0, t1]** ledurile sunt aprinse sau nu

```
void
test_double_locked_leds_at_time(u32 time, bool_ expected_led0, bool_ expected_led1)
{
    for (;;) //while(1) mai frumos scris
    {
        blink(&leds);
        if (have_passed(timer_, time)) //daca pun fix 100 e loterie cand merge cand
nu merge
        {
            TEST_ASSERT_TRUE((digitalRead_(LED0) == expected_led0) &&
            digitalRead_(LED1) == expected_led1);
            return;
        }
    }
}

void
test_double_locked_leds_interval(u32 time_start, u32 time_end, bool_ expected_led0,
bool_ expected_led1)
{
    for (;;)
    {
        blink(&leds);
        if (have_passed(timer_, time_start) && !have_passed(timer_, time_end))
        {
            if (digitalRead_(LED0) != expected_led0 || digitalRead_(LED1) !=
expected_led1)
            {
                TEST_ASSERT_TRUE(FALSE);
                return;
            }
        }
        if (have_passed(timer_, time_end + 1))
        {
            TEST_ASSERT_TRUE(TRUE);
            return;
        }
    }
}
```

Apoi testele pe care le-am rulat sunt urmatoarele:

```
```

```
int

main()
{
    //fac testele la 299ms de exemplu in loc de 300ms fiindca exista
    //posibilitatea ca have_passed(300) sa se activeze intre 2 instructiuni
    //consecutive de aprindere si ASSERT si uneori ar da PASS, alteori nu
    UNITY_BEGIN();
    RUN_TEST(test_leds_double_locked_after_99ms_leds_on);
    RUN_TEST(test_leds_double_locked_after_101ms_leds_on);
    RUN_TEST(test_leds_double_locked_after_500ms_leds_off);
    RUN_TEST(test_leds_double_locked_from_1ms_to_299ms_leds_on);
    RUN_TEST(test_leds_double_locked_from_301ms_to_999ms_leds_off);
    RUN_TEST(test_leds_double_locked_from_1001ms_to_1299ms_leds_on);
    RUN_TEST(test_leds_double_locked_after_1500ms_leds_off);
    RUN_TEST(test_leds_double_locked_after_2099ms_leds_on);
    RUN_TEST(test_leds_double_locked_after_3_cycles_done_leds_off);
    return UNITY_END();
}
```

Fac testele, de exemplu, la 299ms in loc de 300ms fiindca exista posibilitatea ca have_passed(300) sa se activeze intre 2 instructiuni consecutive, cea de blink si cea de ASSERT. Adica blink-ul (si oprirea led-urilor, fiindca dupa 300ms se opresc ledurile)

poate sa se fi intamplat ori exact inaintea ASSERT-ului si atunci e bine, testul trece, ori s-ar fi intamplat la iteratia exact urmatoare ASSERT-ului si atunci nu se mai executa oricum findca testul pica. Si atunci e loterie, uneori trece testul cu 300ms, uneori pica.

Primele 2 teste au scos in afara un **bug** in program **(**pe care l-am vazut cand imi picau alte teste pe care le-am pus initial).

Vedem rezultatul celor 2 teste (doar ele rulate):

```
Processing test_leds in native environment
-------------------------------------------------------------------------------------------------------------------
Building...
Testing...
test\test_leds.cpp:39: test_leds_double_locked_after_99ms_leds_on: Expected TRUE Was FALSE      [FAILED]
test\test_leds.cpp:136: test_leds_double_locked_after_101ms_leds_on     [PASSED]
Program received signal CTRL_BREAK_EVENT (None)
------------------------------ native:test_leds [ERRORED] Took 3.12 seconds ------------------------------

================================================== SUMMARY ==================================================
Environment    Test       Status    Duration
-------------  ---------  --------  ------------
native         test_leds  ERRORED   00:00:03.122
```

Primul test pica: deci dupa 99 de ms, ledurile nu sunt ON,

dar dupa 101 ms ele se fac ON, fiindca al doilea test trece.

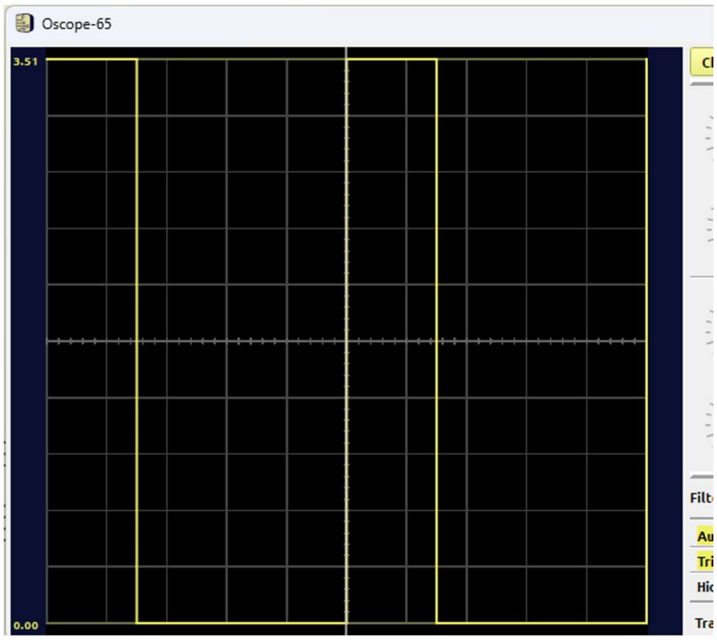
Problema este vizibila cand inspectam cu atentie codul:

```
void
blink(blinking_leds* leds)
{
    if (leds->cycles >= leds->nr_blinks)
    {
        return;
    }
    if (have_passed(leds->led_timer, 100))
    {
        reset_timer(&(leds->led_timer));
        if (leds->count == 0)
        {
            digitalWrite_(leds->pin0, HIGH);
            digitalWrite_(leds->pin1, HIGH);
        }
        else if (leds->count  ==  3)
        {
            digitalWrite_(leds->pin0, LOW);
            digitalWrite_(leds->pin1, LOW);
        }
        leds->count++;
        if (leds->count == 10)
        {
            leds->count = 0;
            leds->cycles++;
        }
    }
}
```

Prima actiune de **digitalWrite_()** in leduri se executa in if-ul de **have_passed(leds->led_timer, 100).** Adica la 100 de ms de cand s-a resetat timerul, si timerul se reseteaza si pe **reset_leds(),** dar mai important pe **init_leds()** pe care il apelam in **setUp().**

Asadar, **BUG:** pentru primele 100 de ms nu se schimba starea ledurilor.

Acest **bug** a fost si in proiectul trecut in modul **LANE_CHANGE** de unde am refolosit codul si nu ne-am fi putut da seama din simulator.



Noi am verificat in simulator, doar **FU = 30%** si numaul de blink-uri sa fie **corespunzator modului in care ne aflam.**

Dar acest bug era insesizabil, fiindca led-urile **fac bine patternul** doar ca cu **100 de ms mai tarziu decat trebuie si** 100 de ms sunt

insesizabile pentru noi in simulator. (a fost greu si sa testam apasarea butonului la mai putin de 100ms).

Una dintre solutii este sa mutam if-urile care schimba starea led-urilor in afara if-ului mare:

```
void
blink(blinking_leds* leds)
{
    if (leds->cycles >= leds->nr_blinks)
    {
        return;
    }
    if (leds->count == 0)
    {
        digitalWrite_(leds->pin0, HIGH);
        digitalWrite_(leds->pin1, HIGH);
    }
    if (leds->count  ==  3)
    {
        digitalWrite_(leds->pin0, LOW);
        digitalWrite_(leds->pin1, LOW);
    }
    if (have_passed(leds->led_timer, 100))
    {
        reset_timer(&(leds->led_timer));
        leds->count++;
        if (leds->count == 10)
        {
            leds->count = 0;
            leds->cycles++;
        }
    }
}
```

Se observa ca trec toate testele:

```
-------------------------------------------------------------------------------
Building...
Testing...
test\test_leds.cpp:135: test_leds_double_locked_after_99ms_leds_on      [PASSED]
test\test_leds.cpp:136: test_leds_double_locked_after_101ms_leds_on     [PASSED]
test\test_leds.cpp:137: test_leds_double_locked_after_500ms_leds_off    [PASSED]
test\test_leds.cpp:138: test_leds_double_locked_from_1ms_to_299ms_leds_on       [PASSED]
test\test_leds.cpp:139: test_leds_double_locked_from_301ms_to_999ms_leds_off    [PASSED]
test\test_leds.cpp:140: test_leds_double_locked_from_1001ms_to_1299ms_leds_on   [PASSED]
test\test_leds.cpp:141: test_leds_double_locked_after_1500ms_leds_off   [PASSED]
test\test_leds.cpp:142: test_leds_double_locked_after_2099ms_leds_on    [PASSED]
test\test_leds.cpp:143: test_leds_double_locked_after_3_cycles_done_leds_off    [PASSED]
------------------------------- native:test_leds [PASSED] Took 17.81 seconds -------------------------------

================================================ SUMMARY ================================================
Environment    Test       Status    Duration
-------------  ---------  --------  ------------
native         test_leds  PASSED    00:00:17.805
================================== 9 test cases: 9 succeeded in 00:00:17.805 ===================================
```

Rularea testelor dureaza mult (**17.81s**) fiindca la noi mock-ul pentru **millis_()** e chiar o functie care lucreaza cu **ms.**

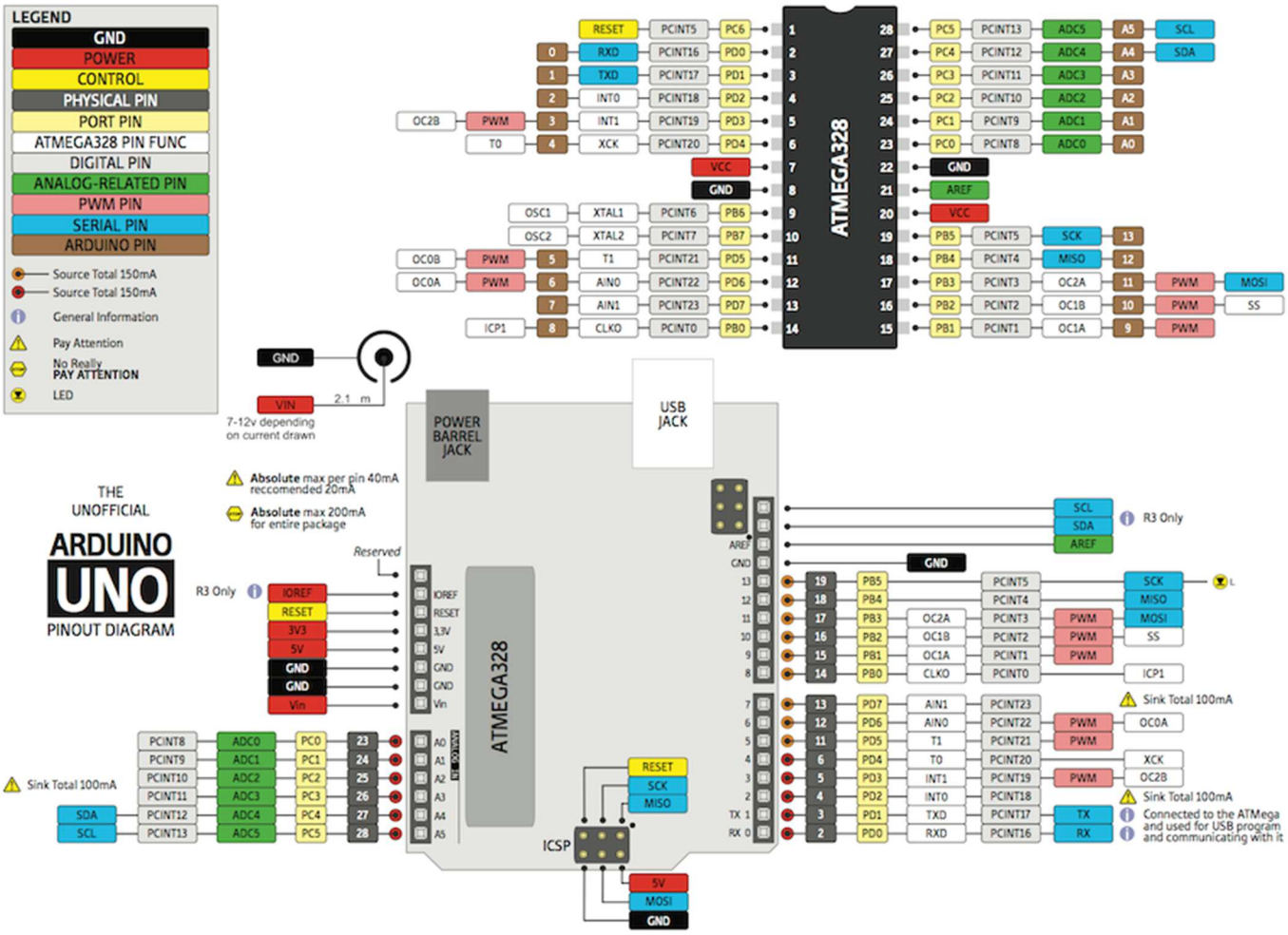
De exemplu testul care vrea sa vada **daca dupa 2099ms led-urle sunt ON**, chiar asteapta **2 secunde.**

Si testul final care verifica ca dupa timpul alocat celor 3 blink-uri (3 secunde), led-urile sunt stinse verifica ca in intervalul 3001ms si 4000ms sunt tot timpul stinse, ceea ce ia **4 secunde.**

Pe viitor am putea schimba de exemplu sa returneze in loc de **ms**, sa zicem **100us**, sau **10us** si asta ne-ar duce la timpi de test foarte mici, buni pentru **testare automata on save** sa zicem. (pentru testarea ledurilor in care functia **blink()** consuma putin timp s-ar putea)

Obs: Nu am ales sa testez modulul de led-uri intentionat fiindca stiam ca are un bug si voiam sa arat ca testarea chiar a rezolvat ceva. M-am gandit ce modul are nevoie de cele mai putine chestii de **mock**-uit ca sa pot sa scriu testele mai repede si s-a intamplat ca chiar era un bug in functia de **blink()** a led-urilor.

4. Implementarea functiilor de arduino cu registrii AVR.



Diferenta este ca **pinMode()**, **digitalRead/Write()** lucreaza cu numarul pinilor de la placa Arduino de la 0 la 13 si A0 ... A5 care sunt macro-uri pentru 14...19), iar registrii lucreaza pinii procesorului din interior ATMega328.

digitalWrite(A2, HIGH) ⇔ setbit(PORTC, 2)

Solutia e sa **mapam pinii exteriori** (ai placii Arduino UNO), la **portul si pozitia** in port din **AVR.**

```
void
locate_pin (IN_PARAM u8 id, OUT_PARAM port_* port, OUT_PARAM u8* nr)
{
    if (id < 8) //PD0-PD7
    {
        *port = P_D;
        *nr = id;
        return;
    }
    id -= 8;
    if (id < 6) //PB0 - PB5
    {
        *port = P_B;
        *nr = id;
        return;
    }
    id -= 6;
    if (id < 6) //PC0 - PC5
    {
        *port = P_C;
        *nr = id;
        return;
    }
    *port = (port_)(-1); //EROARE
    *nr = 8; //EROARE
}
```

Cu aceasta mapare, **digitalWrite()** e o **scriere in PORT**, **digitalRead()** e o citire din **PIN** si pinMode() e o scriere in **DDR**.

```
void
pinMode_(u8 pin, u8 mode)
{
    port_ port; u8 nr;
    locate_pin(pin, &port, &nr);
    switch(port)
    {
        case P_C: {setbit(DDRC, nr, mode);} break;
        case P_D: {setbit(DDRD, nr, mode);} break;
        case P_B: {setbit(DDRB, nr, mode);} break;
        default: break;
    }
}
```

Implementarea functiei millis():

O idee ar fi sa configurezi timerul sa cicleze o data la 1ms si in loop() sa incrementezi un counter in momentul cand OCF0A din TIFR0 e setat. (suntem pe modul CTC) :testbit(TIFR0, OCF0A)

Problema cu pollingul este ca daca stam mai mult de **1ms in loop()** putem sa **ratam un clock** de exemplu:

|----------| = 1ms

//----------// = t_while(1)

|-------//---|----------|--//--------|

Se vede cum daca t_while(1) > 1ms ratam o ciclare de timer, deci ratam o ms.

Solutia e sa folosim intreruperi:

```
void
timer0_init(void)
{
    // p = prescaler N =OCR  k = de cate ori incrementam in intrerupere
    // ca sa obtinem 1 ms
    //   1ms = N * p * k * T_clk
    //     => T_clk * 1ms = 16Mhz * 1ms = 16 000 =
    //     N * p * k = 2^4 * 2^3 * 5^3 = 2^7 * 5^3
    //     alegem p = 2^6 = 64 si N = 2 * 5^3 = 250, k = 1

    TCCR0A = 0b00000010; //OC-urile disconnected si WGM pe CTC
    TCCR0B = 0b00000011; //p = 64, WGM pe CTC, FOC nu mai stiu ce sunt si nu le folosim
    OCR0A = 250 - 1;
}

static volatile u32 cnt = 0;
```

```
ISR(TIMER0_COMPA_vect)
{
    cnt++;
}
```

Millis_() doar returneaza valoarea **cnt** si **are grija de un caz special**.

Cnt e pe 4 octeti, deci e nevoie de 4 instructuni in cod masina de move din Flash (variabila statica **cnt** e cel mai probabil in flash).

Problema e cand ISR-ul se activeaza intre cele 4 instructiuni si avem unii bytes din versiunea veche (inainte de incrementare) si restul de bytes din versiunea noua.

De exemplu, daca initial cnt = **0x000000FF** (pe 4 bytes). Noi apucam sa punem doar ultimul octet in ram si vine intreruperea care face cnt++, cnt ajunge **0x00000100**. acum punem si urmatorii octeti si nr format va fi o combinatie : **0x000001FF.**

Solutia 1. oprim intreruperile, facem citirea, pornim intreruperile

cli()

citim cnt

sei()

Obs: Noi nu pierdem intreruperile care vin cat timp noi le-am demascat (oprit) doar ca se activeaza dupa ce le activam si noi (demascam) adica dupa ce am citit tot **cnt**.

Solutia 2. facem 2 citiri consecutive, adica la distanta de mai putin de 1ms intre ele, o intrerupere poate sa imi corupa doar una dintre citiri, si mai mult poate sa o corupa doar modificand valoarea reala in sus (la una mai mare), deci minimul dintre cele 2 valori va fi valoarea corecta.

```
u32
millis_(void)
{
    static bool_ init = FALSE;
    if (!init)
    {
        init = TRUE;
        timer0_init();
        setbit(TIMSK0, OCIE0A, 1);
        return 0;
    }

    u32 val1 = cnt;
    u32 val2 = cnt;
    return (val1 < val2) ? val1 : val2;
}
```

Obs: Acum ca ma gandesc, nu sunt sigur daca ordinea in cod masina de copiere a bytes-ilor este tot timpul de la LSB la MSB, s-ar putea sa depinda de endian. Daca e de la MSB la LSB, nu mai merge schema, fiindca valoarea corupta o sa fie tot timpul mai mica

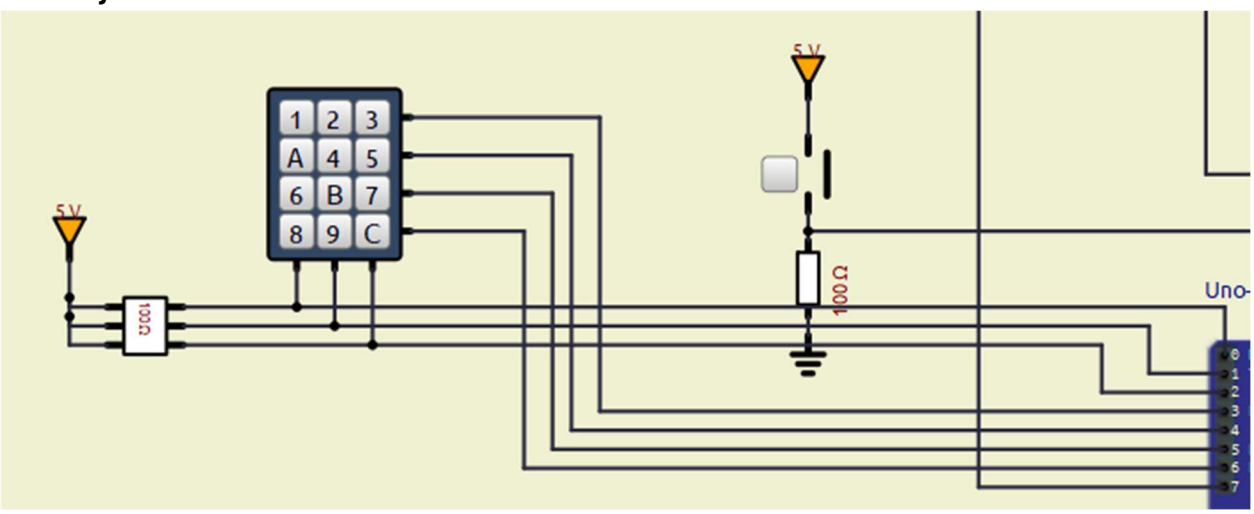
5.Modulul de securitate: keypad scan + keypad / button debounce

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**, 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.

6.Cyclic routine: rularea taskului o data la 20ms

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.