

**UNIVERSITATEA POLITEHNICĂ DIN BUCUREȘTI**  
**FACULTATEA DE ELECTRONICĂ, TELECOMUNICAȚII ȘI**  
**TEHNOLOGIA INFORMAȚIEI**

## **PROIECT 2 – ETAPA I**

**SISTEM DE CONTORIZARE ȘI DE AFIȘARE A**  
**CONSUMULUI DE ENERGIE ELECTRICĂ**

**STUDENȚI:**

LEFTERACHE ALEXANDRU-GABRIEL 431A

CRISTESCU ANA-IOANA 433D

MARINESCU AURORA-CRISTINA 433D

OPREA LIVIA-DANIELA-MIHAELA 433D

**PROFESOR COORDONATOR:**

ZOICAN SORIN

**2022-2023**

# CUPRINS

I. PREZENTAREA TEMEI .....	3
II. DESCRIEREA PRELUCRĂRILOR - GRAFURI ȘI ORGANIGrame .....	4
• ATmega164 .....	4
• ADSP 2181 .....	17
III. SCHEMA BLOC – SCHEMELE ELECTRICE PENTRU PLĂCILE DE EVALUARE.....	23
• ATmega164 .....	23
• EXTENSIA IO (ADSP).....	27
IV. PROIECTAREA DE PRINCIPIU A SCHEMEI CE COMPLETEAZĂ PLACA DE EVALUARE .....	28
V. REZULTATELE SIMULĂRII.....	29
• ATmega164 .....	29
• ADSP 2181 .....	38
VI. COD SURSĂ .....	40
• ATmega164 .....	40
• ADSP 2181 .....	52

# I. PREZENTAREA TEMEI

Tema constă în realizarea unui sistem care contorizează și afișează consumul de energie electrică și este compus din două subsisteme (AVR și DSP). Arhitectura acestora este reprezentată în figura 1:

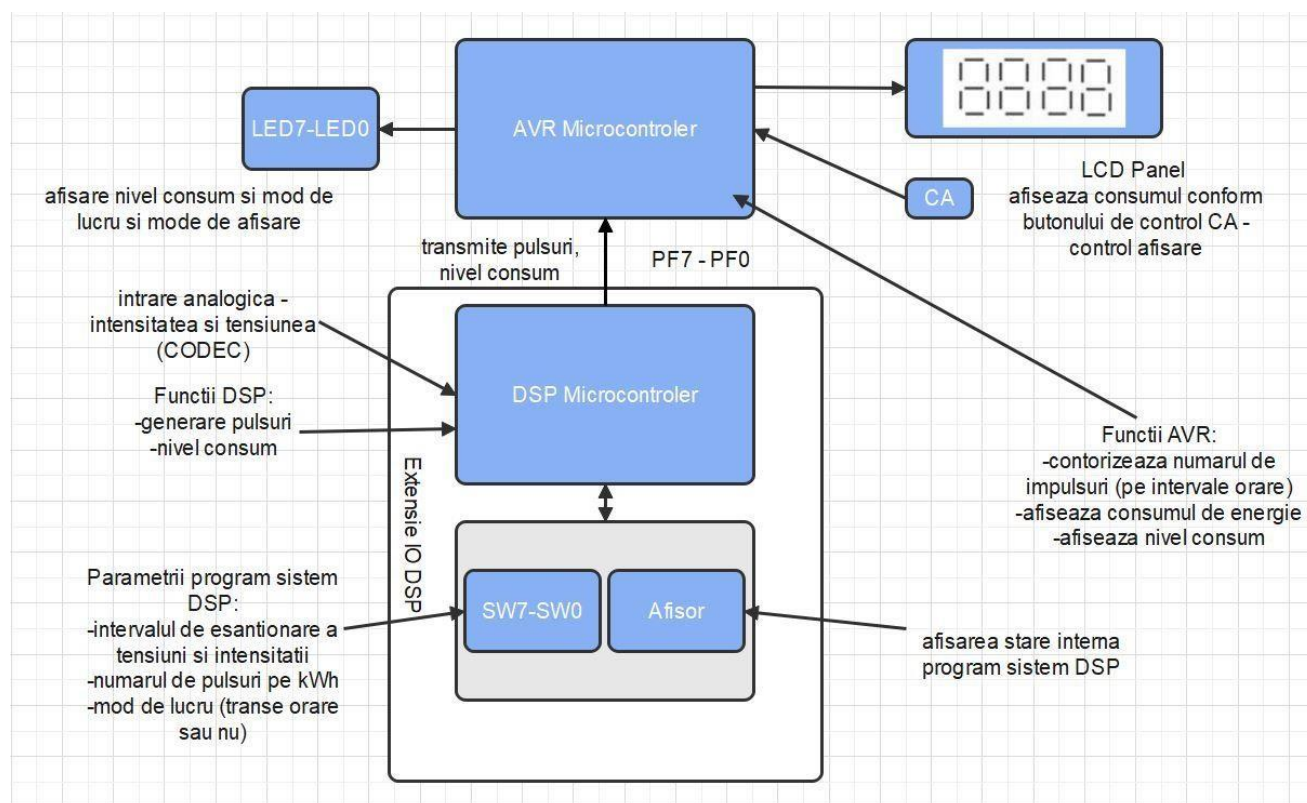


Figura 1 – Schema de ansamblu

Subsistemul DSP are în componența sa placa de evaluare EZ-Kit LITE ADSP2181 și o interfață de intrare-ieșire, IO DSP. Acesta realizează măsurarea tensiunii și intensității, calcularea energiei consumate și generarea unui Pulsurile generate se transmit pe pinul 0 al portului PF, iar puterea consumată pe restul pinilor ai aceluiași port.

Se consideră puterea maximă de 10 kW. Intervalul de eșantionare a tensiunii și intensității  $\Delta T$  și numărul de pulsuri  $P$  per kWh, se vor stabili din SW7-SW0. În cadrul temei, s-a stabilit un mod de lucru al contorizării energiei cu/fără intervale orare.

Subsistemul AVR utilizează un microcontroler ATMega164. Acesta realizează contorizarea pulsurilor, afișarea și calcularea energiei consumate conform modului indicat de subsistemul DSP.

Implementarea la nivel hardware: subsistemul AVR (cu microcontroler ATMega164) și extensia IO DSP

Implementarea la nivel software: descrierea formală a programelor pentru subsistemele AVR și DSP, scrierea codului pentru cele 2 subsisteme ( în limbaj de programare C pentru AVR și în limbaj de asamblare ADSP2181 pentru subsistemul DSP) și testarea programelor în CVAVR și ASTUDIO, respectiv în Visual DSP++ 3.5.

În final se va verifica funcționalitatea sistemului fizic realizat.

## II. DESCRIEREA PRELUCRĂRILOR - GRAFURI ȘI ORGANIGrame

### • ATmega164

Programul principal AVR, reprezentat în organigrama din figura 2 de mai jos, conține următoarele blocuri:

**Inițializări**, unde sunt inițializate variabilele globale, tebelele pentru circuitele logice combinaționale (CLC), tabelele de semnale relevante de la circuitele logice secvențiale (CLS);

**Afișează consum**, având corespondentul DisplayConsumption în codul ce va urma, reprezintă funcția care va face afișarea consumului, ca mai apoi, să se aștepte **Înteruperi** - acest procedeu se realizează în bucla while a programului principal – main – astfel: de fiecare dată când procesorul trece prin bucla while, se afișează câte un digit, așa încât, ținând cont de frecvența mare a procesorului, ochiul uman va fi ”păcălit”, afișajul părând a fi continuu. Practic, el afișează pe rând, dar foarte repede. Dacă apare întrerupere, se trece în rutina de servire a întreruperii, la **SCI**, iar dacă nu apare o întrerupere, procesul se reia.

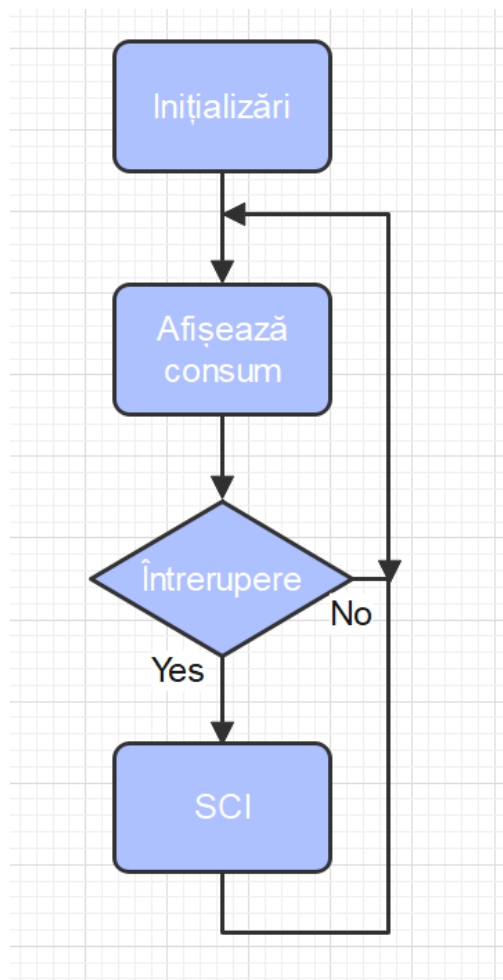
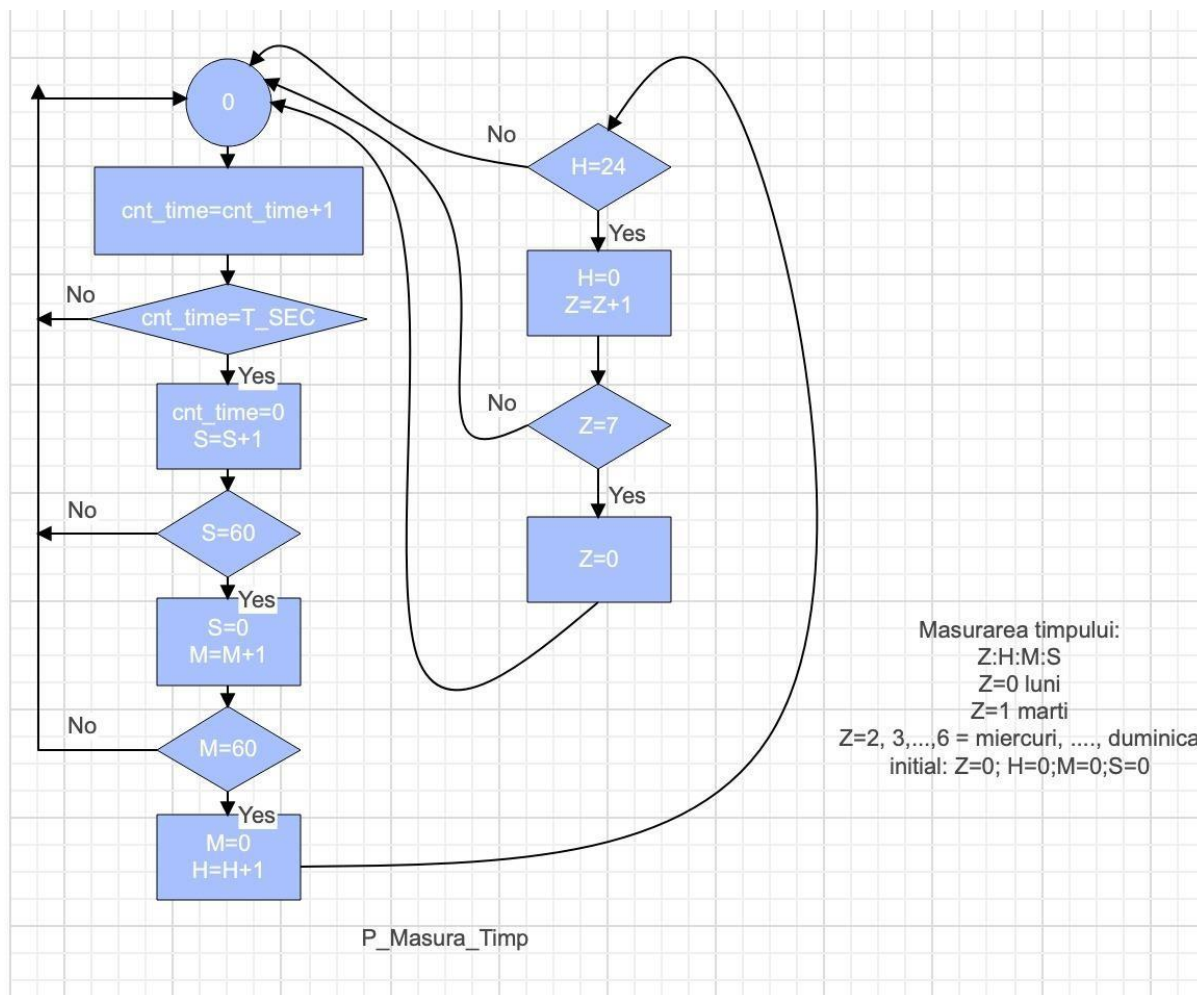


Figura 2 – Organigramă asociată programului principal

## CONTORIZAREA TIMPULUI

Se va implementa sistemul de contorizare a timpului, în funcția UpdateTime();.

În cadrul procesului secvențial asociat contorizării timpului, cu organigrama în figura 3, vom defini câte o variabilă specifică fiecărei unități de timp:  
H - hour; Z - day; M - minutes; S - seconds;



**Figura 3 – Organigramă contorizare timp**

De asemenea, celelalte variabile din cadrul organigramei au semnificații, după cum urmează:

- cnt\_time - contorul de timp
- T\_SEC - numărul de perioade necesare pentru a acoperi o secundă

Pseudocodul asociat programului pentru contorizarea timpului va începe din starea 0, având contorul cnt\_time=0.

Valorile inițiale ale variabilelor H, Z, M, S pot fi inițializate cu valoarea 0 sau pot fi inițializate cu valorile reale la începutul funcționării.

**Pseudocod:**

Functia UpdateTime;

Se incrementează cnt\_time;

Daca cnt\_time este diferit de T\_SEC, programul da return;

Resetare cnt\_time;

Se incrementează S ( contor secunde );

Daca S este diferit de 60, programul da return;

Resetare S;

Se incrementează M ( contor minute )

Daca M este diferit de 60, programul da return;

Resetare M;

Se incrementează H ( contor ore );

Daca H este diferit de 24, programul da return;

Resetare H;

Se incrementează Z ( contor zile );

Daca Z = 7 , se resetează Z;

Programul da return;

**Implementare în limbaj C:**

```
void UpdateTime(){
    cnt_time += 1; //incrementare contor de timp
    if(cnt_time != T_SEC) return;

    cnt_time = 0; // se reseteaza contorul
    S+=1; //incrementeaza contor secunde

    if(S!=60) return;
    S = 0; //se reseteaza nr de secunde
    M += 1; //incrementeaza contor minute

    if(M!=60) return;
    M = 0;
    H += 1;

    if(H!=24) return;
    H = 0;
    Z += 1;

    if (Z == 7) Z = 0;
    return;
}
```

## AFIȘARE CONSUM:

Se determina consumul ce necesita a fi afisat cu ajutorul functiei DisplayConsumption(); ( fie consumul total, fie cel corespunzator intervalului curent). Se calculeaza si se afiseaza succesiv fiecare cifra, apelandu-se Displaydigit().

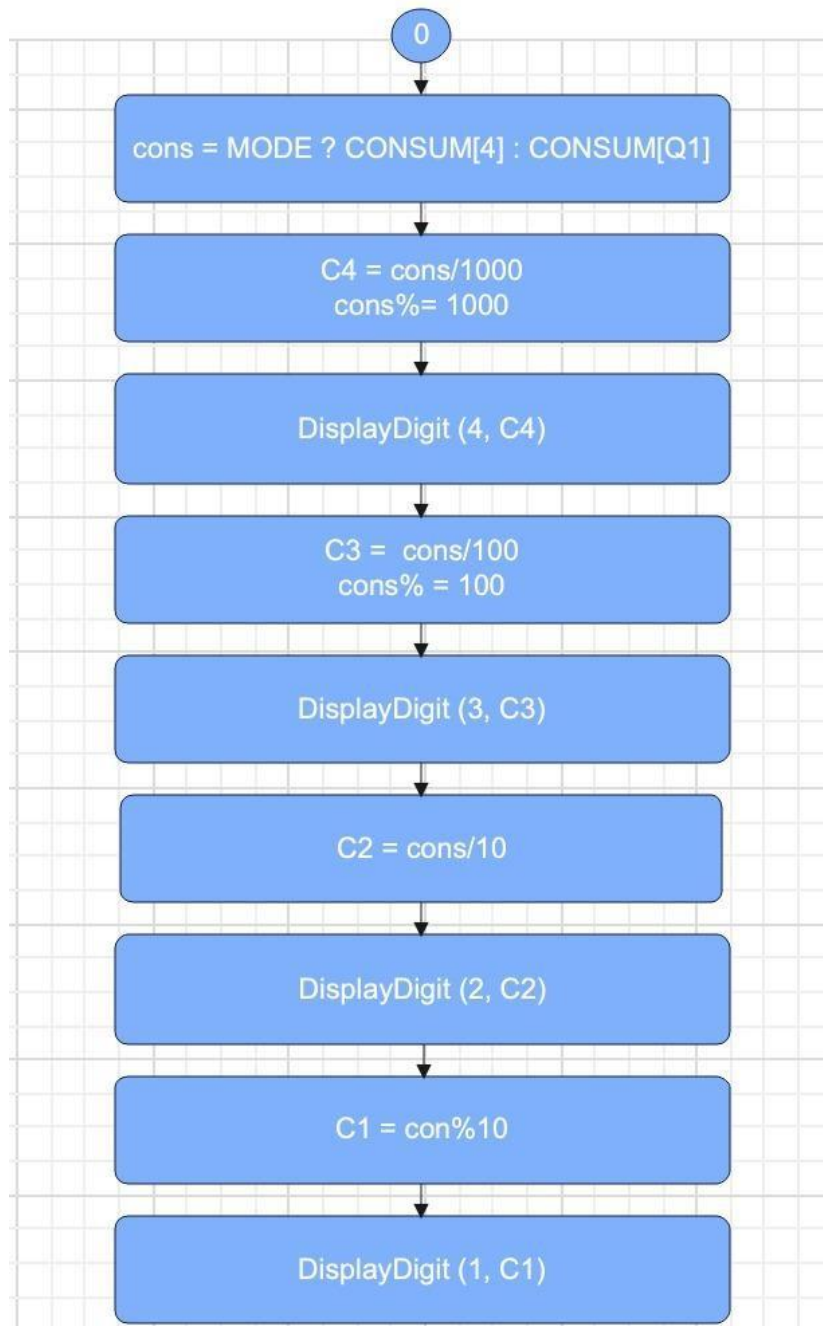


Figura 4 – Organigramă asociată afișării consumului

**Pseudocod:**

```

Functie DisplayConsumption(){

```

```

    Daca MODE este egal cu 1 => se afiseaza consumul total;
    Altfel se afiseaza cifra curenta;

```

```

    C4 = consum impartit la 1000; //cifra miilor
    Consum = restul impartirii la 1000;
    Se apeleaza functia DisplayDigit();
    C3 = consum impartit la 100; //cifra sutelor
    Consum = restul impartirii la 100;
    Se apeleaza functia DisplayDigit();

```

```

    C2 = consum impartit la 10; //cifra zecilor
    Se apeleaza functia DisplayDigit();

```

```

    Consum = restul impartirii la 10; //cifra unitatilor
    Se apeleaza functia DisplayDigit();

```

```

}

```

**Cod impementat în limbaj C:**

```

void DisplayConsumption()

```

```

{
    // We assume:
    // PORTC: PC0 - PC6 -> 7 segments (A-G)
    // PORTD: PD0 - PD3 -> select the common cathode for each digit (multiplexing)
        // PD3 - C4, PD2 - C3, PD1 - C2, PD0 - C1
    // Q - consumption range:
        // 0 -> 00:00 - H1:00
        // 1 -> H1:00 - H2:00      (MON - FRI)
        // 2 -> H2:00 - 00:00 (next day)
        // 3 -> SAT - SUN

```

```

    // The actual approach:
    // Each main loop iteration we multiplex the digits and display one at a time

```

```

    // If CA MODE = 1 -> display total consumption,
    // else -> display consumption based on current range.
    int cons = (MODE) ? CONSUM[4] : CONSUM[Q1];

```

```

    // Compute and display C4
    C4 = cons / 1000;
    cons %= 1000;
    DisplayDigit(4, C4);

```

```

    // Compute and display C3
    C3 = cons / 100;

```



```

cons %= 100;
DisplayDigit(3, C3);
// Compute and display C2
C2 = cons / 10;
DisplayDigit(2, C2);

// Compute and display C1
C1 = cons % 10;
DisplayDigit(1, C1);
}

```

Explicații Pentru metoda DisplayConsumption():

Presupunem port C cu pinii de la 0 la 6, conectați la cele 7 segmente (A-G) ale fiecărui afișor.

Toate segmentele de tipul A vor fi conectate la pinul 0, toate segmentele de tipul B la pinul 1, analog restul.

Pe port D, cu pinii de la 0 la 3, vom face selecția pentru fiecare digit, adică se face conexiunea la masa fiecăruia, ca să validăm respectivul afișor.

Q-ul este indexul intervalului de consum curent.

## AFIȘAREA DIGITULUI:

### Pseudocod:

Functia DisplayDigit{

Bitii de la iesire se fac toti 1.

Exista 4 cazuri:

Caz 4:

// pe iesire = 0x08;

// se iese din caz

Caz 3

// pe iesire = 0x04;

// se iese din caz

Caz 2

// pe iesire = 0x02;

// se iese din caz

Caz 1

// pe iesire = 0x01;

// se iese din caz

Punem iesirea pe PORTD; // selectia afisorului pe 7 segmente

PORTC va lua configuratia binara digitului respectiv.

}

### Cod impementat în limbaj C:

```
void DisplayDigit(char currentDisplay, char digit)
{
    // Select the desired display (turn on the pin
    // corresponding to the desired digit (C4/C3/C2/C1)
    char output = 0xff;
    switch (currentDisplay)
    {
        case 4:
            // Turn PD3 on
            //output &= 0b11110111;
            output = 0x08;
            break;
        case 3:
            // Turn PD2 on
            // output &= 0b1111011;
            output = 0x04;
            break;
        case 2:
            // Turn PD1 on
            output = 0x02;
            break;
        case 1:
            // Turn PD0 on
            output = 0x01;
            break;
    }

    // Assign output to PORTC in order to select the desired display;
    PORTD = output;

    // Set PORTC pins to the corresponding digit
    PORTC = DIGITS[digit];
}
```

Metoda DisplayDigit(), primul parametru pe care ni-l dă ca argument este indexul display-ului pe care dorim să îl afișăm.

C este portul pe care îl conectăm la afișoarele noastre și o să îi dăm valoarea DIGITS(digit) unde DIGITS este tabela noastră CLC (afișează pe afișor luând reprezentarea combinației binare care ne reprezintă cifra pe acesta).

După ce am selectat combinația binară care ne dă cifra pe care am dorit să o afișăm, vom selecta prin multiplexare afișorul pe care dorim noi să facem afișarea.

Vom lua o variabilă output pe care o vom inițializa la 0xff, output=ieșirea pe care o dăm pe port D, ieșirea pe care facem multiplexarea. Vom trece în 0 pinul corespunzător afișorului pe care vrem să facem afișarea.

## ACTUALIZARE CONSUM

Proces secvențial asociat măsurării nivelului de energie. Variabilele utilizate sunt :

- **PULSE** – valoarea pulsului venit din partea ADSP
- **DP** – durata puls
- **CONSUM** – CLC, tabelă ce conține valorile consumului pe intervale orare.
- **Q** – stare interval curent de contorizare

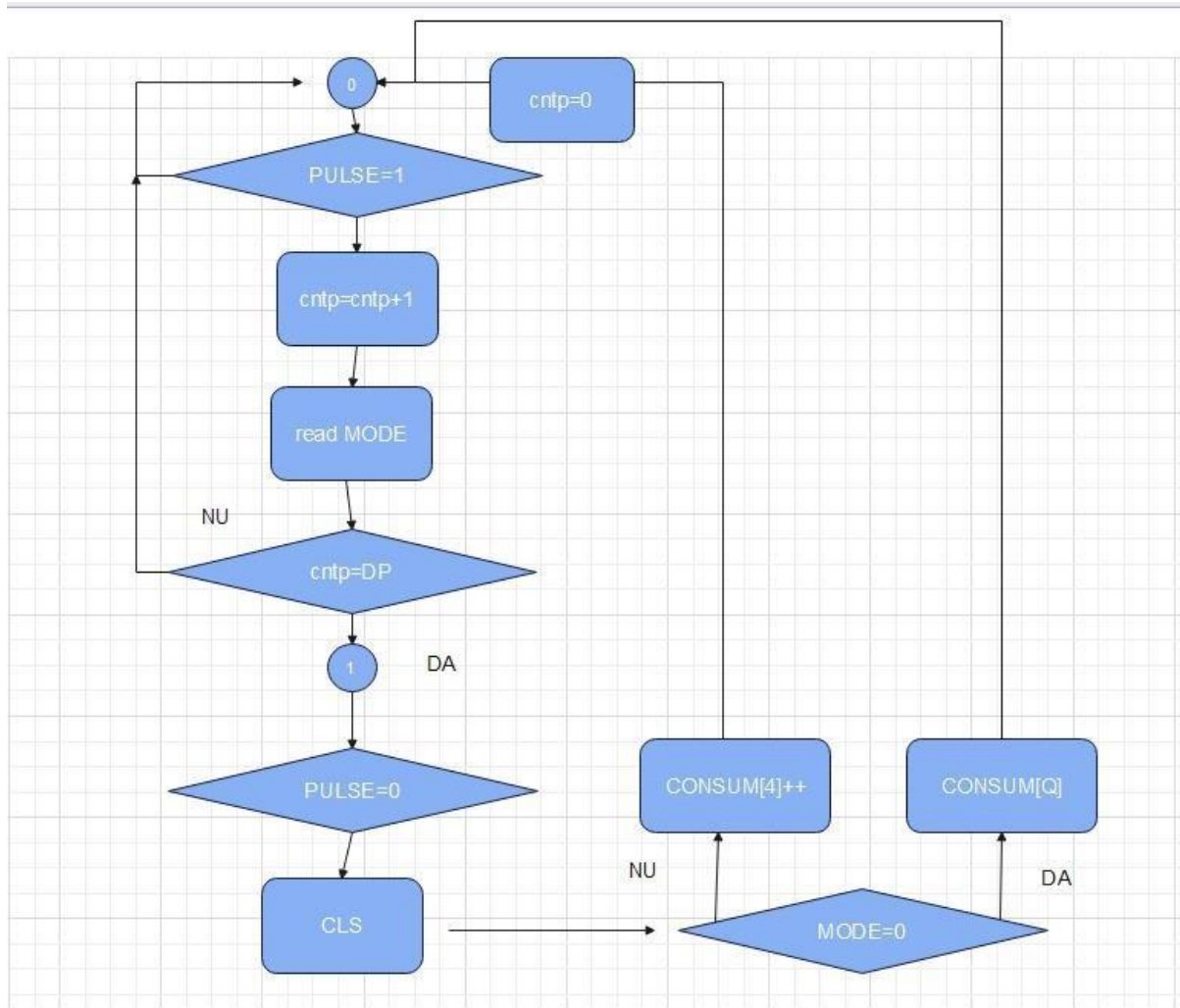


Figura 5 – PS actualizare consum

### Pseudocod:

//Pornim din starea 0 și așteptăm ca PULSE să fie trecut în "1" (primim pulsuri din partea ADSP)

```
void masurare_energie(){
```

Pentru powerLevel, se aplica masca si se shifteaza cu 1 bit.

Cazurile lui S2:

```
{
```

```

caz 0:
{ dacă PULSE = 1

    //Se incrementeaza contor_puls;
    // Daca este primul impuls ( adica modeFlag = 1 ) {
        //se identifica bitul de pe pozitia si se reseteaza modeFlag
    //Daca cntP=DP, atunci S2 devine 1 ( trece in starea urmatoare), altfel S2 = 0;
    }

```

```

Caz 1:
    Daca PULSE=0;
//Se apeleaza functia CLS();
    Daca MODE = 0 , se incrementeaza consumul starii curente
    Altfel se incrementeaza consumul total
// S2 = 0 ; se intoarce in starea initiala
// se resetaza cntP;

```

### **Cod impementat în limbaj C:**

```

void UpdateConsumption()
{
    PowerLevel = (PINA & 0x7E) >> 1;

    switch(S2)
    {
        case 0:
        {
            // If PULSE is on, start counting
            if (PULSE)
            {
                // Increment cntP
                cntP += 1;

                // Reset reading flag
                // PORTD &= 0x7f;

                if (modeFlag)
                {
                    MODE = (PINA & 0x80) >> 7;
                    modeFlag = 0;
                }

                // Go further if the pulse period has passed,
                // otherwise go back wait for sensding ack again.
                S2 = (cntP == DP) ? 1 : 0;
            }
            break;
        }
        case 1:

```

```

{
  if (~PULSE)
  {
    // Update current consumption range
    CLS();

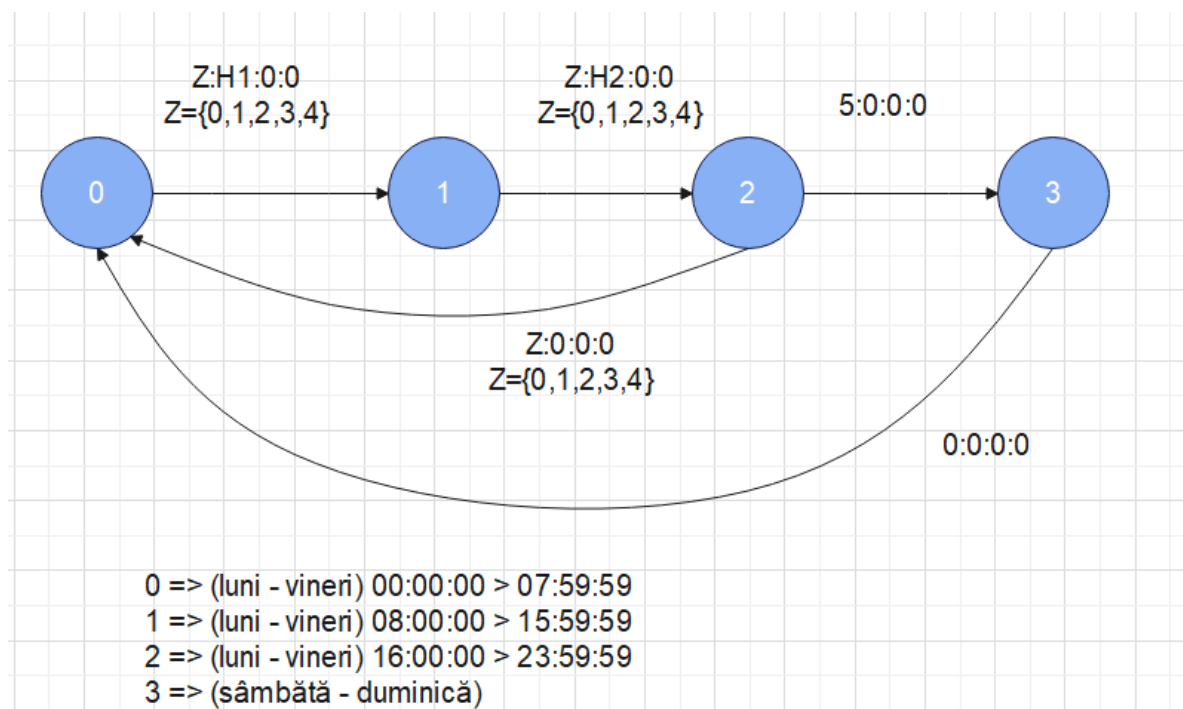
    // Increment consumption
    if (MODE == 0)
    {
      CONSUM[Q] += 1; // Working range on
    }
    else
    {
      CONSUM[4] += 1; // Working range off
    }

    // Wait for another pulse
    S2 = 0;
    cntP = 0;
  }
  break;
}
}

```

## DETERMINARE INTERVAL DE ÎNREGISTRARE

**Graf asociat CLS:**



**Figura 6 – Graf asociat CLS interval orar**

Graful CLS asociat determinării intervalului curent de timp este prezentat în figura de mai sus. Există patru stări aferente celor patru intervale orare. În graf, orele sunt marcate cu 0:00, H1, H2.

Pentru o mai bună înțelegere a funcționalității, vom prezenta un exemplu concret, considerând  $H1 = 8$ , respectiv  $H2 = 16$ . Astfel, pentru starea "0" corespundătoare zilelor din timpul săptămânii, de luni până vineri, ne referim la intervalul orar cuprins între 0:00:00 și 07:59:59. Starea următoare, "1", corespunde tot zilelor din timpul săptămânii și face trecerea de la ora 08:00:00 la ora 15:59:59. Starea "2" face trecerea de la ora 16:00:00 la ora 23:59:59, tot în timpul săptămânii. Starea "3" corespunde finalului de săptămână, sâmbătă și duminică.

### **Pseudocod:**

```
functia CLS(){
    out = iesirea
    now = momentul curent de timp
    adr = ADR(TAB[Q])
    ready = 0 (conditie)
    i = 0 (index)

    atata timp cat ready=0{
        daca(now = adresa de la indexul i){
            Q = adr[i+1]
            ready = 1
        }
        altfel daca (adr[index i] = terminator) ready = 1
        altfel i = i+2
    }
    out = Tout[Q]
}
```

### **Cod impementat în limbaj C:**

```
void CLS()
{
    long int now = (Z<<24) | (H<<16) | (M<<8) | S;

    long int *adr = TABA[Q];
    char ready = 0;
    int i = 0;
    long int out = 0;

    while (!ready)
    {
        if (now == adr[i]) {
            S1 = adr[i + 1];
            ready = 1; // Stop iterating through while
        }
        else if (adr[i] == T) ready = 1;
        else i = i+2;
    }
}
```

## AFIȘARE NIVEL PUTERE

Cunoaștem din datele problemei nivelul maxim de putere: 10kW. Prin urmare, împărțim intervalul 0 – 10kW în trei posibile subintervale în care se poate regăsi puterea consumată la un moment dat de timp. Nivelul puterii (exprimat în kW) este primit paralel pe 8 biți, alături de pulsul de contorizare a consumului, pe pinii portului A, dinspre ADSP.

### Organigramă asociată:

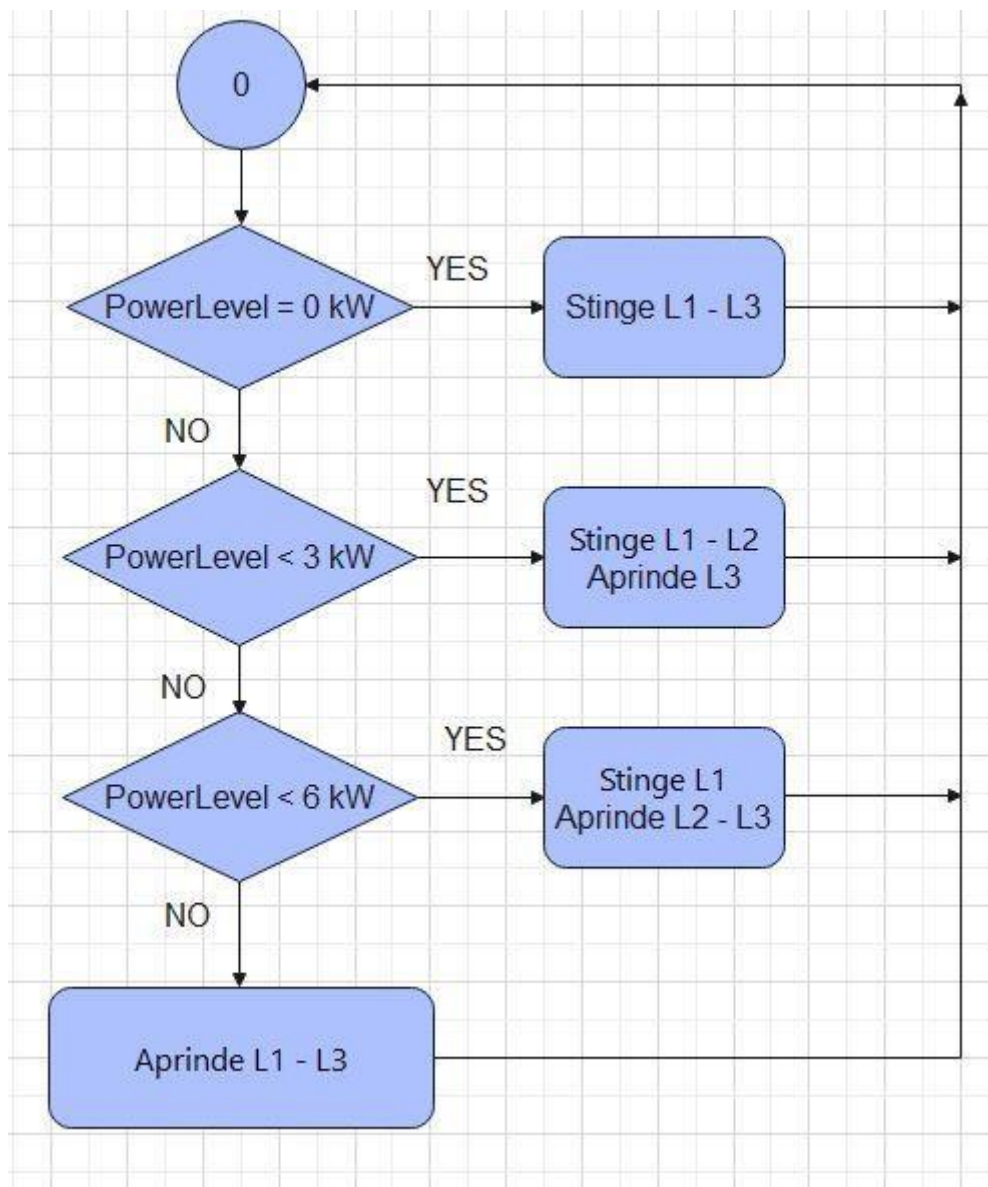


Figura 7 – Organigramă asociată afișării nivelului de putere

### Pseudocod:

Funcție AfișarePutere(){

Dacă PowerLevel = 0  
    Stinge LED 1-3

```
Altfel dacă PowerLevel < 3
    Sting LED 1-2
    Aprinde LED 3
```

```
Altfel dacă PowerLevel < 6
    Sting LED 1
    Aprinde LED 2-3
```

```
Altfel
    Aprinde LED 1-3
}
```

### **Cod impementat în limbaj C:**

```
void DisplayPowerLevel()
{
    char out;

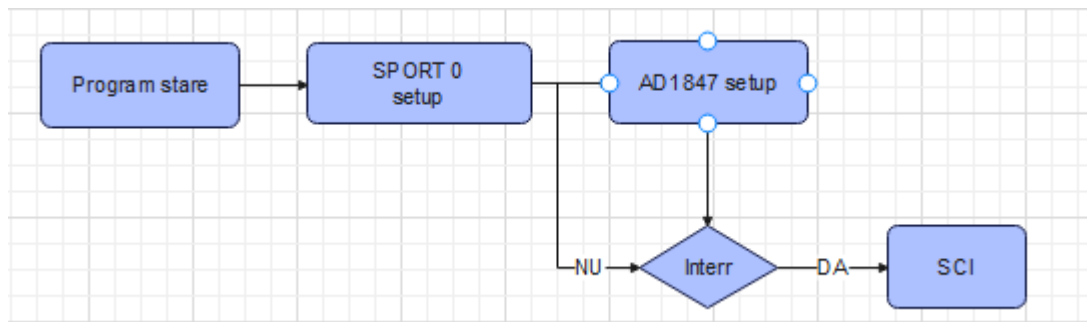
    if (!PowerLevel)      // PowerLevel = 0 kW
    {
        out = CLC_LEVEL[0];
    }
    else if (PowerLevel < 3) // 0 < PowerLevel < 3 kW
    {
        out = CLC_LEVEL[1];
    }
    else if (PowerLevel < 6) // 3 <= PowerLevel < 6 kW
    {
        out = CLC_LEVEL[2];
    }
    else                  // PowerLvel >= 6 kW
    {
        out = CLC_LEVEL[3];
    }

    // Delete PB7-PB5
    PORTB &= 0x1f;

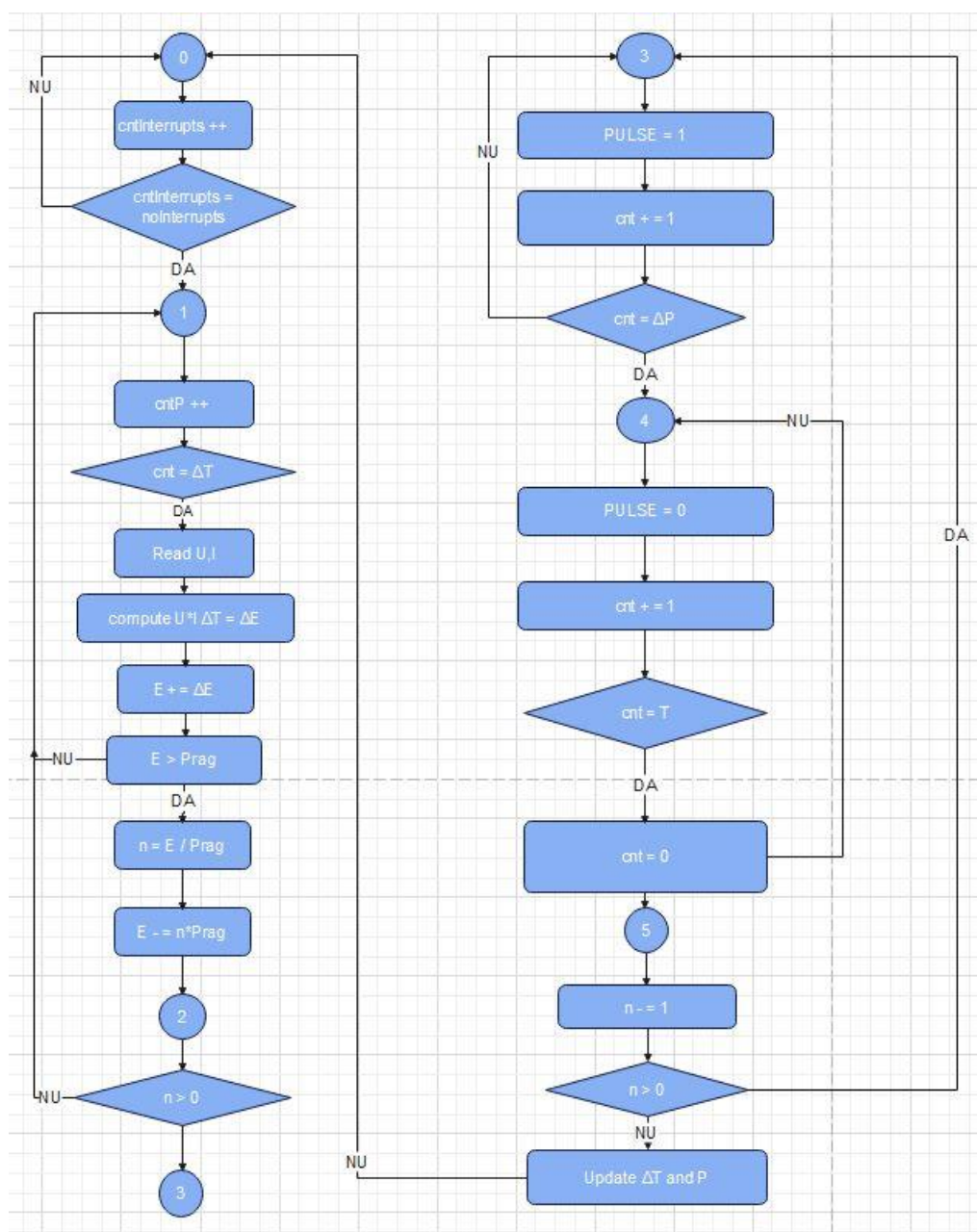
    // Display out on PB7-PB5
    PORTB |= out;
}
```



- **ADSP 2181**



**Figura 8a – Organigramă program principal (ADSP 2181)**



**Figura 8b – Rutina de servire a întreruperii (SCI)**

Simularea întreruperii externe (IRQ2) se realizează la fiecare 1000 de cicluri executate. Aceasta se regăsește în codul sursă sub forma etichetei intitulată SCI. În cadrul acestei secvențe este implementată funcționarea procesorului sub forma unui proces secvențial, ilustrat anterior.

Semnificația variabilelor utilizate :

- **cntInterrupts** – contor întreruperi
- **noInterrupts** – număr întreruperi necesar pentru acoperirea unui interval de 20ms.
- **cntP** – contor durată de eșantionare
- **dT** – număr de contorizări în starea 0 necesar pentru acoperirea intervalului de eșantionare
- **dE** – energie consumată pe intervalul de eșantionare
- **Prag** – referit în codul sursă drept "Threshold" reprezintă pragul de energie la care se efectuează transmiterea unui puls. (E.g: dacă stabilim să transmitem 4 pulsuri/kWh, pragul de transmitere pentru un puls va fi de 250Wh).
- **n** – număr de impulsuri de trimis
- **PULSE** – valoarea palierului pulsului
- **DP** – număr contorizări corespunzătoare duratei palierului de 1 logic
- **Cnt** – contor corespunzător perioadei impulsului.

#### Implementarea rutinei de servire a întreruperii în limbaj de asamblare :

**sci:**

```
ena sec_reg;
ay0 = dm(Q);           // Read current state
ar = PASS ay0;         // ar = 0 + Q
if eq jump Q0;         // If ar = Q = 0 => jump towards Q0

ar = ay0 - 1;          // ar = Q - 1
if eq jump Q1;         // If ar = 0 => jump towards Q1

ax0 = 2;
ar = ay0 - ax0;        // ar = Q - 2
if eq jump Q2;         // If ar = 0 => jump towards Q2

ax0 = 3;
ar = ay0 - ax0;        // ar = Q - 3
if eq jump Q3;         // If ar = 0 => jump towards Q3

ax0 = 4;
ar = ay0 - ax0;        // ar = Q - 4
if eq jump Q4;         // If ar = 0 => jump towards Q4

ax0 = 5;
ar = ay0 - ax0;        // ar = Q - 5
if eq jump Q5;         // If ar = 0 => jump towards Q5
rti;
```

**Q0:**

// Implementing a counter in order to sync with ATmega

```

// (which has timing interrupts at every 20ms;
ay0 = dm(cntInterrupts);      // Read current cycles counter
ar = ay0 + 1;                  // Increment the cycles counter
dm(cntInterrupts) = ar;
ax0 = ar;
ay0 = dm(noInterrupts);       // Get the necessary cycles number
af = ax0 - ay0;                // Compute cntInterrupts -
noInterruptsif eq jump GO_Q0;  // If not equal => return
    ax1 = 1;
dm(Q) = ax1;
rti;

GO_Q0:
ax1 = 0;
dm(Q) = ax1;
rti;
////////////////////////////////////

//////////////////// Q = 1 //////////////////////

Q1:
// Check if the sampling period is complete
ay1 = dm(cntP);                // ay1 = cntP
ar = ay1 + 1;                  // ar = cntP + 1
dm(cntP) = ar;                 // cntP = ar = cntP + 1
ax1 = dm(dT);                  // ax1 = dT
ay1 = dm(cntP);                // Refresh: ay1 = cntP (incremented)
ar = ax1 - ay1;                // ar = dT - cntP
if gt rti;                      // if dT > cntP => return
ax1 = 0;
dm(cntP) = ax1;                // Restart counting

ax1 = 0;                        // Write PULSE = 0 at 0xFF
IO(PORT_OUT) = ax1;

    // Read U & I
mx0 = dm (rx_buf + 2); // Citeste senzorii de tensiune & curent
my0 = dm (rx_buf + 1);

mx0 = 200;                      // U
my0 = 10;                       // I

// Compute dE
mr = mx0 * my0 (uu); // mr = U * I
dm(P) = mr0;                // P (power) = U * I
mx1 = dm(dT);                // mx1 = dT
my1 = mr0;                    // my1 = U * I
mr0 = dm(E);                 // mr = E
mr1 = dm(E + 1);
mr = mr + mx1 * my1 (uu);    // mr = E + U * I * dT = E + dE

```

```

dm(E) = mr0;                                // Save E = E'
dm(E + 1) = mr1;

ax0 = dm(E);
ax1 = dm(E + 1);

ay0 = dm(Threshold);
ay1 = dm(Threshold + 1);

DIS AR_SAT;
    ar = ax0 - ay0;
    ar = ax1 - ay1 + C - 1, ax0 = ar;
    ax1 = ar;
if lt rti;

// COMPUTE_N:
si = 1;
DIVIDE:
// ax1 ax0 -> E
// ay1 ay0 -> TH
// Save last iteration's result:
dm(E) = ax0;
dm(E + 1) = ax1;

DIS AR_SAT;
    ar = ax0 - ay0;
    ar = ax1 - ay1 + C - 1, ax0 = ar;
    ax1 = ar;
if lt jump STOP;
ar = si;
ar = ar + 1;
si = ar;
jump DIVIDE;

STOP:
dm(n) = si;
ax1 = 2;
dm(Q) = ax1;                                // Q = 2;
rti;

////////////////////////////////////

////////// Q = 2 //////////
Q2:
    ax0 = 0;
    ax1 = 3;
    ay0 = 0;
        ay1 = dm(n);                        // ay0 = n
    af = PASS ay1;                          // ar = n
    if le ar = ax0 + ay0;                    // if n <= 0 => return to Q0

```

```

    if gt ar = ax1 + ay0;                // else => return to Q1
    dm(Q) = ar;                          // Go to Q = 2
    rti;
    //////////////////////////////////////

    ////////////////////////////////////// Q = 3 //////////////////////////////////////
Q3:
    // Generating the pulse //
    // ax1 = 1;                      // Set PULSE = 1
    // ay1 = sr0;
    // ar = ax1 or ay1;

    ax1 = 1;
    // dm(Prog_Flag_Data) = ax1; // PF = PPPP PPP1 (PULSE = 1)
    IO(PORT_OUT) = ax1;           // Write PULSE = 1 at 0xFF
    ay1 = dm(cntG);
    ar = ay1 + 1;                 // Increment cnt
    dm(cntG) = ar;
    ax1 = ar;                     // ax1 = cnt
    ay1 = DP;                     // ay1 = DP
    af = ax1 - ay1;
    ar = 3;                       // Next state => default 3
    if eq ar = ar + 1; // If cntG = DP => Go to Q4
    dm(Q) = ar;                   // Set Q state value
    rti;
    //////////////////////////////////////

    ////////////////////////////////////// Q = 4 //////////////////////////////////////
Q4:
    ax1 = 0;                      // Write PULSE = 0 at 0xFF
    IO(PORT_OUT) = ax1;
    ay1 = dm(cntG);
    ar = ay1 + 1;                 // Increment cnt
    dm(cntG) = ar;
    ax1 = T;
    ay1 = ar;
    af = ax1 - ay1;              // Check whether cnt = T
    ar = 4;
    if eq ar = ar + 1;           // If so, go to Q = 4
    dm(Q) = ar;
    rti;
    //////////////////////////////////////

    ////////////////////////////////////// Q = 5 //////////////////////////////////////
Q5:
    ax0 = 0;
    dm(cntG) = ax0;
    ay1 = dm(n);
    ar = ay1 - 1;
    if gt jump GO_TO_Q3;

```

```
// if le jump UPDATE_INFO;
ax0 = 0;
dm(Q) = ax0;
rti;

GO_TO_Q3:
ax0 = 3;
dm(Q) = ax0;    // Switch to Q = 3
rti;
////////////////////////////////////
```

### III. SCHEMA BLOC – SCHEMELE ELECTRICE PENTRU PLĂCILE DE EVALUARE

#### • ATmega164

Schema bloc a sistemului de contorizare și afișare a consumului de energie electrică este prezentată în figura de mai jos:

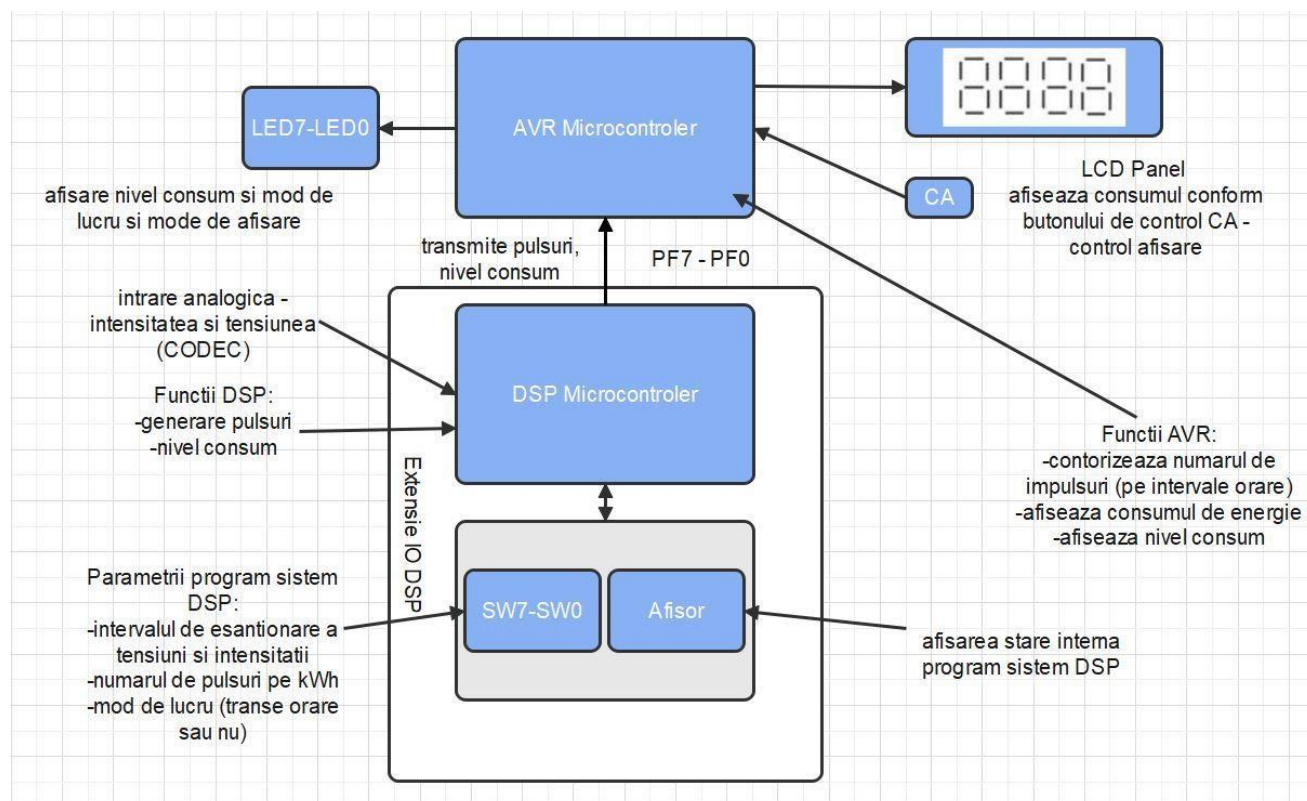


Figura 9 – Schemă bloc

#### DESCRIEREA PIESELOR PRIMARE, IN CONTEXTUL MICROCONTROLLER-ULUI:

##### 1. PoR – Power on Reset

Este un semicircuit format dintr-un resistor în serie cu un condensator (R1+C1), care generează un semnal de “reset”, care se transmite în întregul sistem, asigurând faptul că microcontroller-ul va porni în aceeași condiție de fiecare dată când este alimentat prin intermediul condensatorului.

Când tensiunea de alimentare este aplicată circuitului, condensatorul C1 începe să se încarce prin intermediul rezistorului R1, iar tensiunea de pe condensator crește lent de la 0V la tensiunea de alimentare. În timpul acestei încărcări, semnalul de reset este în stare activă (adesea, nivelul logic LOW), asigurând că microcontroller-ul nu va începe să execute codul

din program până când tensiunea de pe condensator nu atinge o anumită valoare. Odată ce tensiunea de pe condensator atinge această valoare, semnalul de reset devine inactiv (nivelul logic HIGH), iar microcontroler-ul începe să execute codul din program.

Alegerea valorilor: dacă valoarea rezistorului este prea mică, atunci curentul de încărcare al condensatorului va fi mare, ceea ce poate afecta negativ consumul de curent al sistemului. Pe de altă parte, dacă valoarea rezistorului este prea mare, atunci încărcarea condensatorului va dura mai mult timp, ceea ce poate afecta timpul de reset. În mod obișnuit, valoarea rezistorului R1 este aleasă între 4,7 k $\Omega$  și 10 k $\Omega$ .

Valoarea condensatorului C1 este aleasă în funcție de cerințele de timp de reset și de tensiunea de alimentare. Dacă valoarea condensatorului este prea mică, atunci timpul de încărcare a acestuia va fi scurt, ceea ce poate duce la un reset instabil. Pe de altă parte, dacă valoarea condensatorului este prea mare, atunci timpul de încărcare va fi lung, ceea ce poate afecta timpul de reset. În mod obișnuit, valoarea condensatorului C1 este aleasă între 0,1  $\mu$ F și 10  $\mu$ F, iar valoarea de 1  $\mu$ F este o valoare comună și potrivită pentru multe aplicații.

Concluzie PoR: acest reset este necesar pentru a asigura pornirea în bune condiții a procesorului; în lipsa lui, tensiunile tranzitorii care apar în momentul alimentării pot duce procesorul într-o stare incertă.

## **2. Cristalul cu cuarț (C2 și C3)**

Cristalul de cuarț X, împreună cu condensatoarele C2, C3 și cu amplificatorul intern de la bornele XTAL1,2 formează un oscilator cu cuarț. Aceste componente trebuie să se afle, de asemenea, cât mai aproape de pinii respectivi ai procesorului.

Pinii XTAL1 și XTAL2 de la microcontroller-ul Atmega164 sunt utilizați pentru a conecta un oscilator cu cristal de cuarț la microcontroller. În acest mod, semnalul oscilatorului cu cristal este utilizat pentru a sincroniza diferitele operații ale microcontroller-ului, cum ar fi execuția codului sau comunicarea prin interfața serială.

Pentru a forma oscilatorul cu cristal, se utilizează un cristal de cuarț, care are proprietatea de a oscila la o frecvență specifică atunci când este supus unei tensiuni electrice. Acest cristal este conectat la bornele XTAL1 și XTAL2 ale microcontroller-ului, iar între aceste borne sunt conectate două condensatoare, C2 și C3.

Condensatoarele C2 și C3 sunt utilizate pentru a stabiliza semnalul de la cristal și pentru a preveni apariția undelor de zgomot și a oscilațiilor nedorite. Aceste condensatoare sunt plasate cât mai aproape posibil de bornele XTAL1 și XTAL2 ale microcontroller-ului pentru a minimiza interferențele și distorsiunile semnalului.

Amplificatorul intern din microcontroller-ul ATmega164 este utilizat pentru a amplifica semnalul oscilatorului cu cristal și pentru a-l folosi ca sursă de sincronizare pentru operațiile microcontroller-ului. Acest amplificator este proiectat pentru a amplifica semnalul de la cristal la nivelurile adecvate pentru a fi utilizat de către microcontroller.

În concluzie, conexiunea oscilatorului cu cristal la microcontroller-ul Atmega164 implică utilizarea unui cristal de cuarț, a două condensatoare pentru stabilizarea semnalului și a unui amplificator intern pentru amplificarea semnalului de la cristal și utilizarea acestuia ca sursă de sincronizare. Aceste componente trebuie să fie plasate cât mai aproape posibil de bornele XTAL1 și XTAL2 ale microcontroller-ului pentru a minimiza interferențele și distorsiunile semnalului.



### 3. R2, D2

În general, atunci când se conectează un LED la ieșirea unui circuit digital fără un tranzistor de comandă, acesta trebuie poziționat invers, adică între pinul circuitului și Vcc, astfel încât să se evite posibilele daune produse de un curent prea mare. În mod normal, când un pin digital este setat la nivel logic 0, acesta oferă o cale de scurgere a curentului către masă, astfel încât LED-ul să nu primească un curent prea mare și să fie protejat.

Cu toate acestea, în cazul procesorului AVR, curentul oferit de un pin digital este destul de constant, indiferent dacă pinul este setat la nivel logic 0 sau 1. Prin urmare, în cazul procesorului AVR, un LED poate fi poziționat în mod normal, adică între pinul digital și masă, fără a fi necesar un tranzistor de comandă. Aceasta înseamnă că LED-ul se va aprinde când pinul este setat la nivel logic 1 și se va stinge când pinul este setat la nivel logic 0.

Rezistența R2 este utilizată pentru a limita curentul prin LED la o valoare sigură, care să nu dăuneze LED-ului sau pinului digital. În cazul de față, rezistența R2 este calculată pentru a limita curentul prin LED la aproximativ 10 mA, ceea ce este o valoare sigură pentru majoritatea LED-urilor.

Prin urmare, în cazul procesorului AVR, nu este necesar să se monteze un LED în mod invers, ci acesta poate fi poziționat normal, între pinul digital și masă, și controlat direct de pinul digital, fără a fi nevoie de un tranzistor de comandă.

### 4. SW1

Butonul SW1 leagă PD5 la masa (0 logic) în momentul apăsării. Întrucât, atunci când nu este apăsat, starea pinului PD5 nu este definită, va trebui activată o rezistență de pull-up intern prin software.

### 5. MOSI/MISO/SCK/RXD/TXD (pot fi utilizate într-o etapă ulterioară)

ATMega164 este un microcontroller AVR cu 16KB de memorie programabilă Flash, care are mai multe porturi periferice, inclusiv porturile MOSI, MISO, SCK, RXD și TXD. Aceste porturi sunt utilizate pentru a conecta microcontrolerul cu alte dispozitive periferice prin intermediul unui protocol de comunicație serială.

Portul MOSI (Master Out Slave In) este utilizat pentru a transmite date de la microcontroler la un dispozitiv periferic. Este conectat la intrarea de date a dispozitivului periferic și este utilizat în general în comunicații SPI.

Portul MISO (Master In Slave Out) este utilizat pentru a primi date de la un dispozitiv periferic către microcontroler. Este conectat la ieșirea de date a dispozitivului periferic și este utilizat în general în comunicații SPI.

Portul SCK (Serial Clock) este utilizat pentru a sincroniza comunicarea între microcontroler și dispozitivul periferic. Este utilizat pentru a stabili viteza de comunicație între dispozitivul periferic și microcontroler și este conectat la intrarea de ceas a dispozitivului periferic.

Portul RXD (Receiver Data) este utilizat pentru a primi date de la un dispozitiv extern la microcontroler. Este utilizat în comunicațiile seriale asincrone (UART).

Portul TXD (Transmitter Data) este utilizat pentru a transmite date de la microcontroler la un dispozitiv extern. Este utilizat în comunicațiile seriale asincrone (UART).

În general, aceste porturi periferice sunt utilizate pentru a permite microcontrolerului să comunice cu alte dispozitive din sistem, precum senzori, module de afișaj sau alte microcontrollere. Acest lucru poate fi utilizat pentru a implementa o gamă largă de aplicații, cum ar fi controlul robotic, senzorii de mediu și multe altele.

## **6. C13**

Într-un microcontroler, conversia analog-digitală (ADC) este utilizată pentru a converti semnalele analogice în semnale digitale, astfel încât acestea pot fi prelucrate de către procesorul digital. Pentru a asigura o conversie analog-digitală precisă, este important să se stabilizeze tensiunea de referință a ADC-ului.

În general, tensiunea de referință este furnizată de o sursă externă sau internă și poate fi afectată de zgomotul de surse externe, variațiile de temperatură sau alte perturbații. Pentru a minimiza acest efect, se poate folosi un condensator de filtrare în paralel cu sursa de tensiune.

În cazul de față, condensatorul C13 este utilizat pentru a filtra tensiunea de referință AREF (VREF ADC), ceea ce poate ajuta la eliminarea zgomotului sau a altor perturbații care pot afecta precizia conversiei analog-digitale. C13 este conectat la portul AREF al microcontrolerului AVR și trebuie să fie dimensionat corect pentru a asigura o performanță optimă a ADC-ului.

În esență, utilizarea condensatorului C13 ajută la stabilizarea tensiunii de referință pentru ADC și poate îmbunătăți precizia conversiei analog-digitale. Este important să se utilizeze un condensator adecvat pentru a asigura performanța corectă a sistemului.

## **7. C5, C6**

Aceste condensatoare sunt numite condensatoare de decuplare și au rolul de a preveni apariția zgomotului de comutare pe liniile de alimentare. Atunci când un circuit digital comută, acesta consumă o cantitate mare de energie electrică într-un timp foarte scurt. Această comutare bruscă poate provoca fluctuații de tensiune pe liniile de alimentare, ceea ce poate afecta funcționarea altor circuite din sistem.

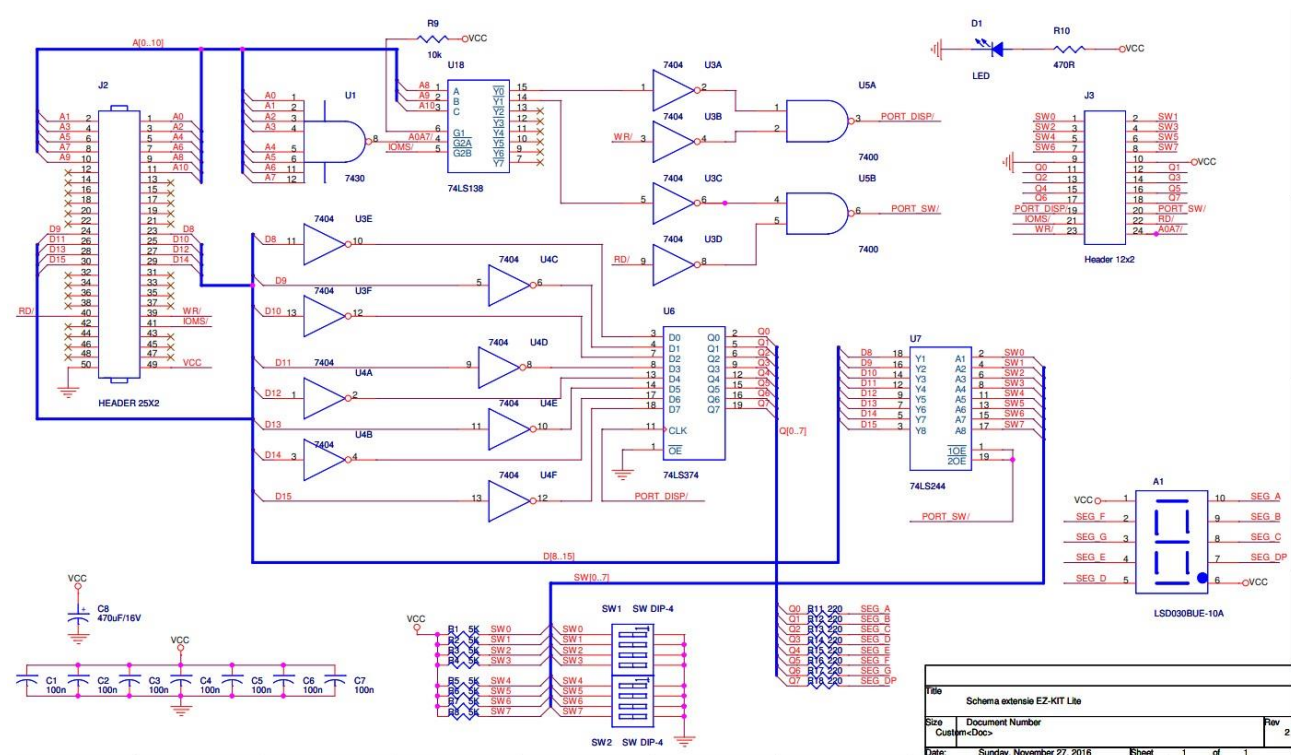
Pentru a preveni această problemă, condensatoarele de decuplare sunt conectate între pinii de alimentare și pământ. Acestea au capacitatea de a stoca o anumită cantitate de energie electrică, care poate fi utilizată pentru a compensa consumul brusc de energie provocat de comutarea circuitului digital.

De asemenea, este important ca aceste condensatoare să fie plasate cât mai aproape de procesor, deoarece orice lungime a traseului dintre condensator și procesor poate crea o impedanță care poate afecta performanța circuitului. Prin plasarea condensatoarelor cât mai aproape de procesor, se minimizează impedanța circuitului și se asigură o alimentare stabilă și curată pentru ATmega164.

## **8. U3**

Pentru alimentare se folosește stabilizatorul U3 cu 3 terminale (7805), 2 condensatoare de 100nF necesare pentru stabilitatea funcționării acestuia, și dioda D1

- **EXTENSIA IO (ADSP)**



### 1) CHIP-ul 74LS138

Pinii G2A/-G2B/ reprezintă intrările de selecție care determină care dintre ieșirile decodorului vor fi activate, atunci când G1 este setat, prin Vcc la nivelul '1' logic. Noi ne folosim doar de pinul G2A/ pe care este transmis rezultatul în urma operației NAND între biții A0-A7. Aceștia sunt transmiși, de la microcontroller pe busul de date și trebuie să fie toți "1", adică 0xFF, ca să se activeze decodorul. Mai departe, pe pinul Y0/ de ieșire se transmite bitul 0 care trece împreună cu bitul semnalului de WR/, printr-o poartă NAND. WR/ este transmis tot de la microcontroller, conexiunea făcându-se prin J3.

## 2) CHIP-ul 74LS374

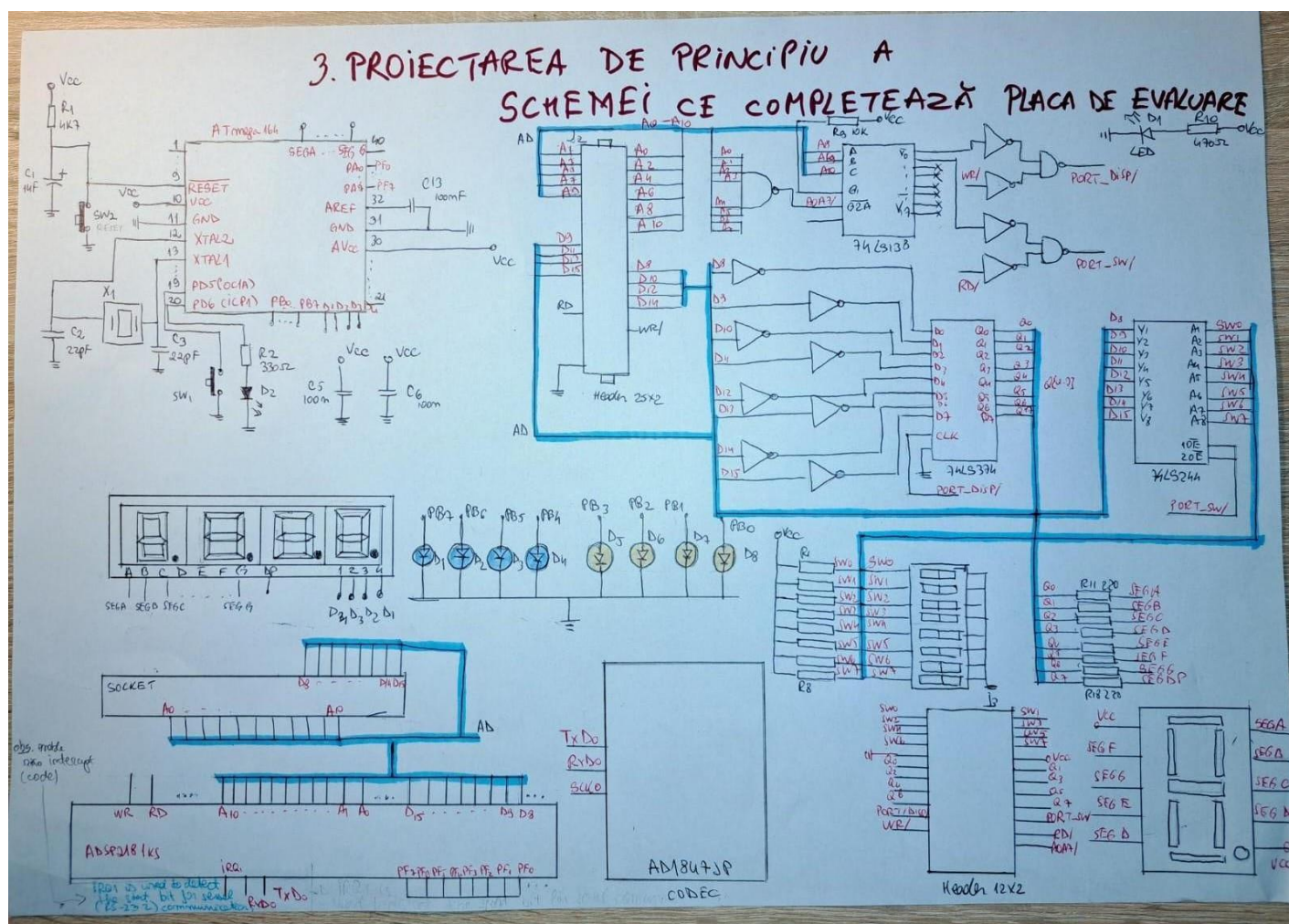
27

### 3) CHIP-ul 74LS244

Buffer Tristate, cu semnal de open-enable(de control), pe 2 biți, activ în 0. Este utilizat pentru a activa/dezactiva ieșirile circuitului integrat. Biții de intrare sunt biții veniti de la switch. Acesta, din urmă, are 2 stări: când este deschis avem "1 logic" (la Vcc, iar rezistențele de 5k, limitează curentul, întrucât aceste chipuri se ard la o intensitate electrică prea mare, iar când este închis , avem "0 logic", care este conectat la masă.

Din 74LS244 se transmit biții de semnal receptionați de la switch, atunci când semnalul PORT\_SW/ este activ, pentru citire. Biții de adrese sunt trimiși mai departe pe busul de date comun (între intrările lui 74LS374 și biții de la microcontroller, conectate prin J2). Acești biți intră în 74LS5374 și cum am menționat anterior, semnalul de ceas PORT\_DISP va detecta ieșirea, implicit afișarea pe ecran.

#### IV. PROIECTAREA DE PRINCIPIU A SCHEMEI CE COMPLETEAZĂ PLACA DE EVALUARE



**Figura 11 – Proiectarea de principiu a schemei complete**

## V. REZULTATELE SIMULĂRII

### • ATmega164

S-a realizat un proiect de test în care am implementat funcția MockPulse() pentru a simula pulsurile venite din partea microcontroller-ului ADSP. Ea este apelată în rutina de servire a întreruperilor. Întreruperile interne sunt generate la fiecare 20ms, ceea ce înseamnă că vor exista 50 de întreruperi/s. Funcția astfel creată va genera un palier de 1 la 50 de întreruperi, adică simulăm un puls/secundă. Practic, în urma utilizării funcției MockPulse() ar trebui să observăm cum sistemul ATmega164 contorizează și actualizează pulsurile venite pe PINA.

### Funcția MockPulse() implementată în limbaj C:

```
void MockPULSE()
{
    switch(S_PULSE)
    {
        case 0:
        {
            cntMockPulse = 0;
            PULSE = 1;
            S_PULSE = 1;
            break;
        }
        case 1:
        {
            cntMockPulse += 1;
            PULSE = 0;
            if (cntMockPulse == 49)
                S_PULSE = 0;
            break;
        }
    }
}
```

Schema realizată în Proteus 8 Professional este prezentată în figura de mai jos :

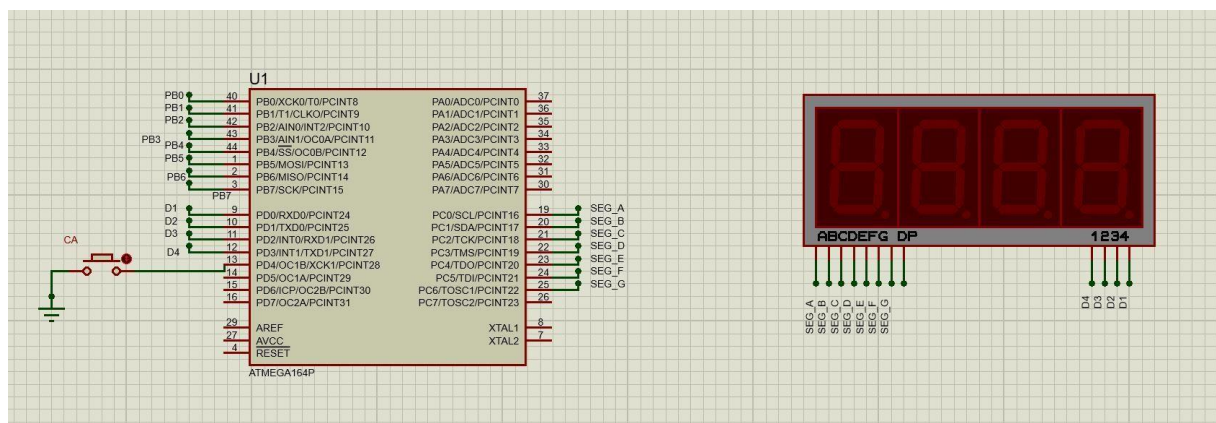


Figura 12 – Atmega164 + Display



Microcontroller-ul ATmega164A are 44 de pini, insuficienți pentru a conecta fiecare segment independent al afișorului. Prin urmare trebuie să realizăm multiplexarea celor patru afișoare ale display-ului. Deși afișarea digiților va fi realizată pe rând, datorită frecvenței mare de lucru a procesorului, impresia utilizatorului va fi aceea că aceștia sunt aprinși simultan.

Pentru testarea multiplexării am legat microcontroller-ul ATmega164 la un display cu 4 afișoare pe 7 segmente astfel :

- Pinul 19 – SEG\_A – PC0;
- Pinul 20 – SEG\_B – PC1;
- Pinul 21 – SEG\_C – PC2;
- Pinul 22 – SEG\_D – PC3;
- Pinul 23 – SEG\_E – PC4;
- Pinul 24 – SEG\_F – PC5;
- Pinul 25 – SEG\_G – PC6;

Exemplificarea funcționării display-ului: terminalul A este comun pentru toate segmentele de tipul A, multiplexarea propriu-zisă între cele 4 afișoare fiind realizată cu ajutorul pinilor 1-4. Selecția afișorului dorit este realizată prin activarea pinului corespunzător. Aceștia sunt conectați la microcontroller prin pinii 9-12 (pe schemă), astfel:

- Pinul 12 (PD0) – D4,
- Pinul 11 (PD1) – D3,
- Pinul 10 (PD2) – D2,
- Pinul 9 (PD3) – D1;

Conform cerinței există două moduri de funcționare a sistemului: pe intervale orare, respectiv fără (consum total). Atât modul de funcționare, cât și intervalele orare sunt reprezentate cu ajutorul LED 4-LED 8, astfel:

- I. LED 4 – Mod de lucru (aprins – contorizare fără intervale orare, stins – contorizare cu intervale orare)
- II. LED 5-6 – Intervalul curent de contorizare.
- III. LED 7-8 – Intervalul de contorizare corespunzător consumului afișat pe ecran. Se va observa cum prin acționarea succesivă a butonului (semnalul CA dat de către utilizator) se poate afișa consumul realizat pe fiecare dintre intervalele orare existente (dacă modul de lucru presupune acest lucru).

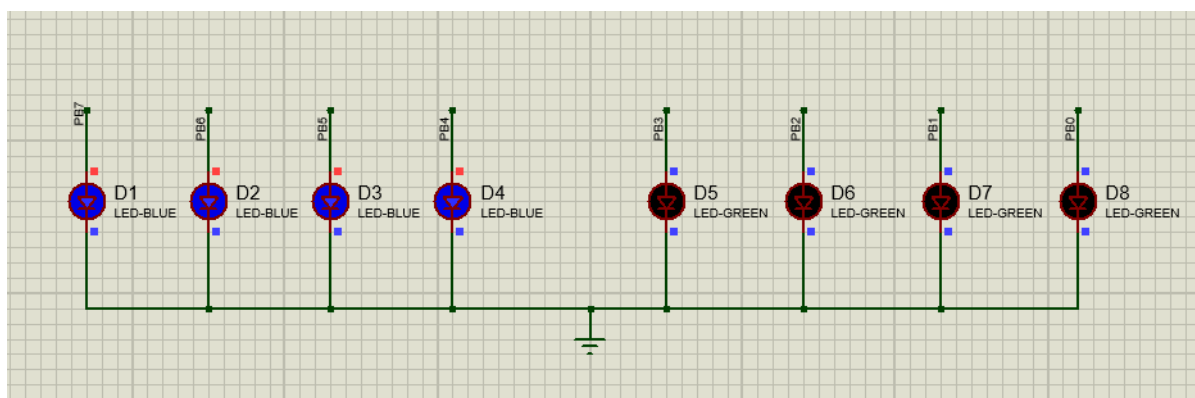
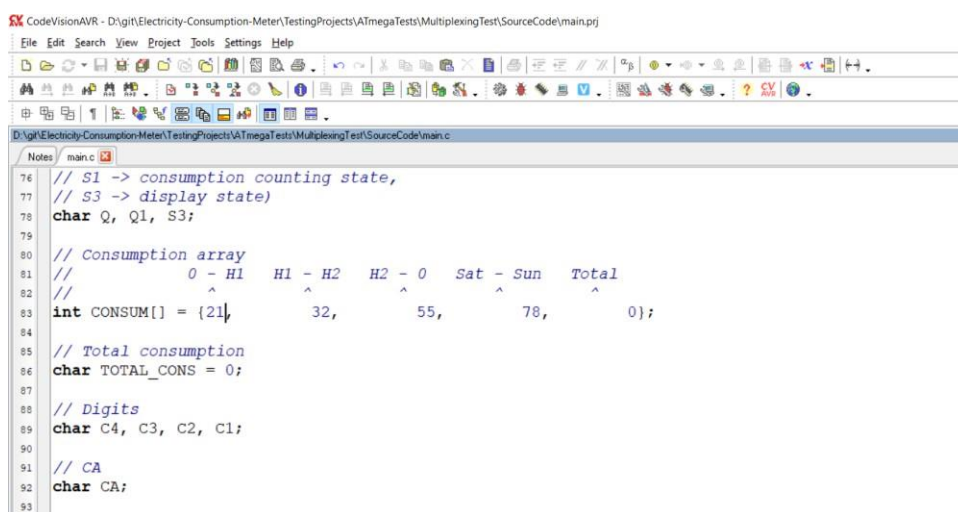


Figura 13 – LED-uri afișare stare AVR

- 1) În modul de lucru pe intervale orare (MODE = 0) distingem 4 intervale de consum:
- Intervalul **00:00 - H1:00** (în zilele lucrătoare) - este reprezentat pe LED-urile de afișare (LED 4-5 sau LED 7-8) prin combinația binară: **01** (unde 0 – LED stins, 1 – LED aprins) .
  - Intervalul **H1:00 - H2:00** (în zilele lucrătoare) - este reprezentat pe LED-urile de afișare (LED 4-5 sau LED 7-8) prin combinația binară: **10** (unde 0 – LED stins, 1 – LED aprins) .
  - Intervalul **H2:00 - 00:00** (în zilele lucrătoare) - este reprezentat pe LED-urile de afișare (LED 4-5 sau LED 7-8) prin combinația binară: **11** (unde 0 – LED stins, 1 – LED aprins) .
  - Intervalul de **week-end** (sâmbătă și duminică) - este reprezentat pe LED-urile de afișare (LED 4-5 sau LED 7-8) prin combinația binară: **00** (unde 0 – LED stins, 1 – LED aprins) .

În simularea acestui mod de lucru LED 4 este stins, în timp ce LED 5 și LED 6 indică modul de consum curent. Prin acționarea butonului CA, se modifică atât combinația LED-urilor 7, respectiv 8 (ce indică intervalul orar pentru care afișăm consumul), cât și consumul reprezentat pe display.

Pentru exemplificarea funcționalității considerăm un proiect de test realizat în CV AVR, în care am introdus manual valorile tabelului intitulat CONSUM.



```
76 // S1 -> consumption counting state,
77 // S3 -> display state)
78 char Q, Q1, S3;
79
80 // Consumption array
81 //      0 - H1   H1 - H2   H2 - 0   Sat - Sun   Total
82 //
83 int CONSUM[] = {21,      32,      55,      78,      0};
84
85 // Total consumption
86 char TOTAL_CONS = 0;
87
88 // Digits
89 char C4, C3, C2, C1;
90
91 // CA
92 char CA;
93
```

Figura 14 – Setarea vectorului de consum

Inițial sistemul se află în starea 0 (luni, intervalul 00:00 – H1:00). Atât LED 5-6, cât și LED 7-8 se află în configurația "01" corespunzătoare acestui mod:

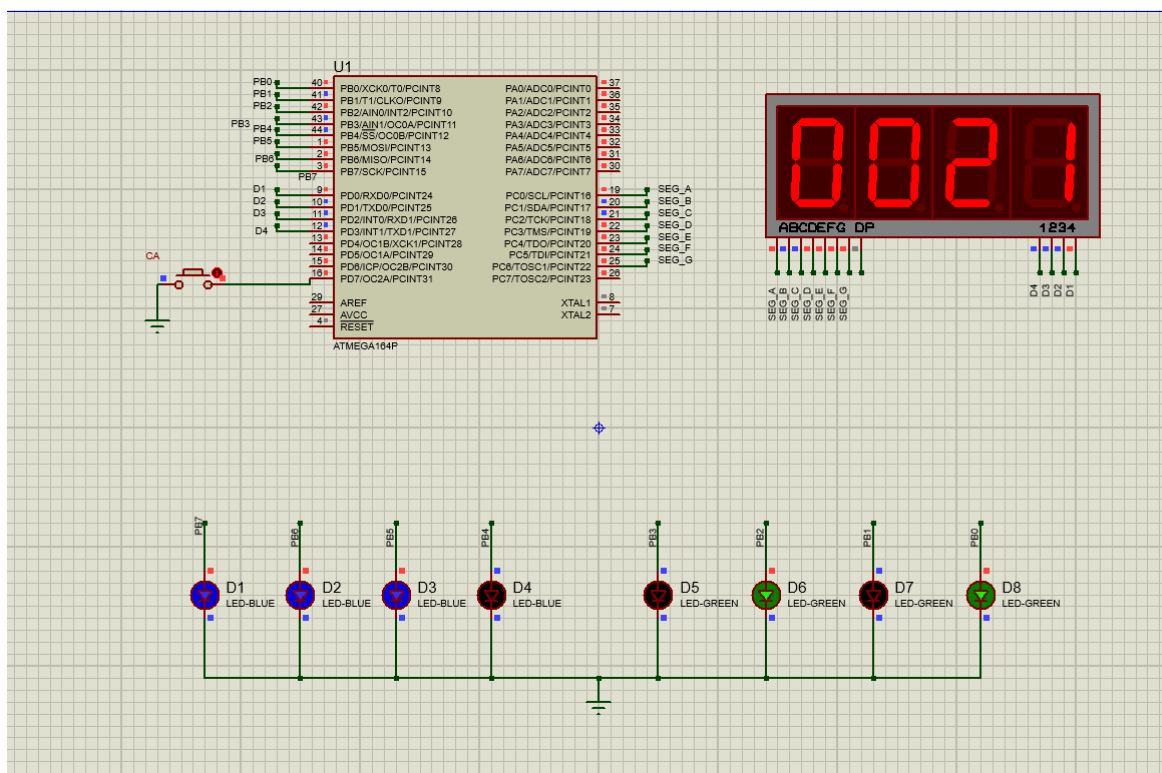


Figura 15 – Stare inițială

Prin apăsarea butonului CA se vor afișa succesiv valorile prezente în tabela CONSUM, însoțite de actualizarea LED-urilor 7, respectiv 8.

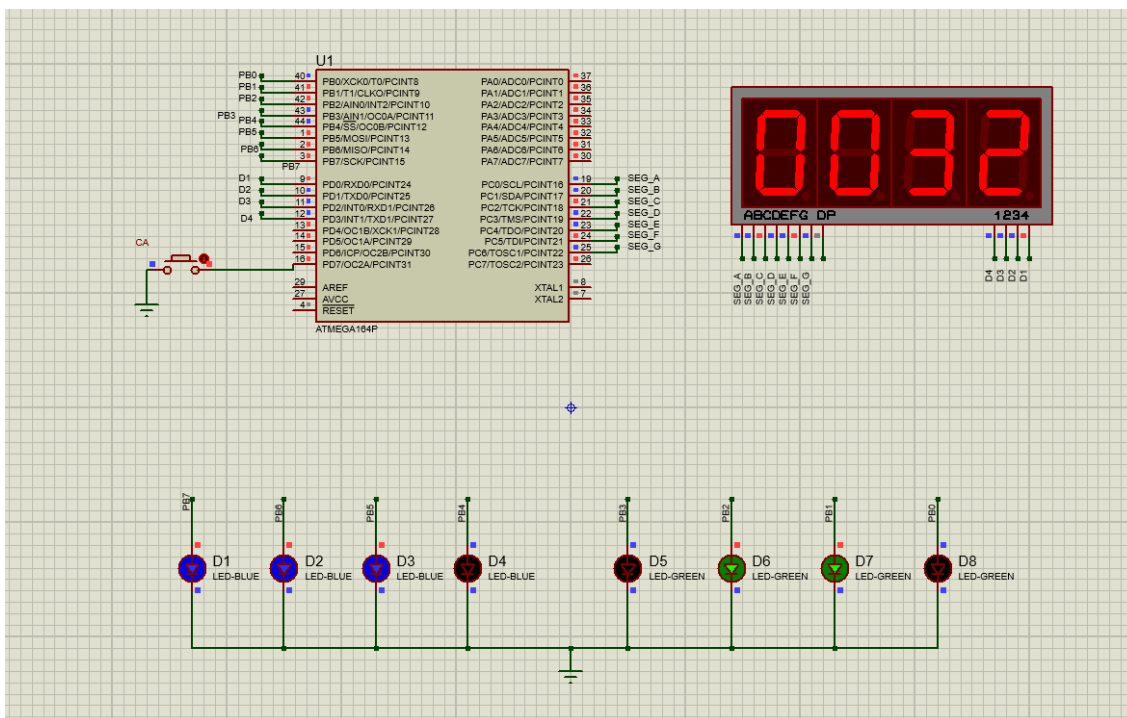


Figura 16 – Stare succesivă



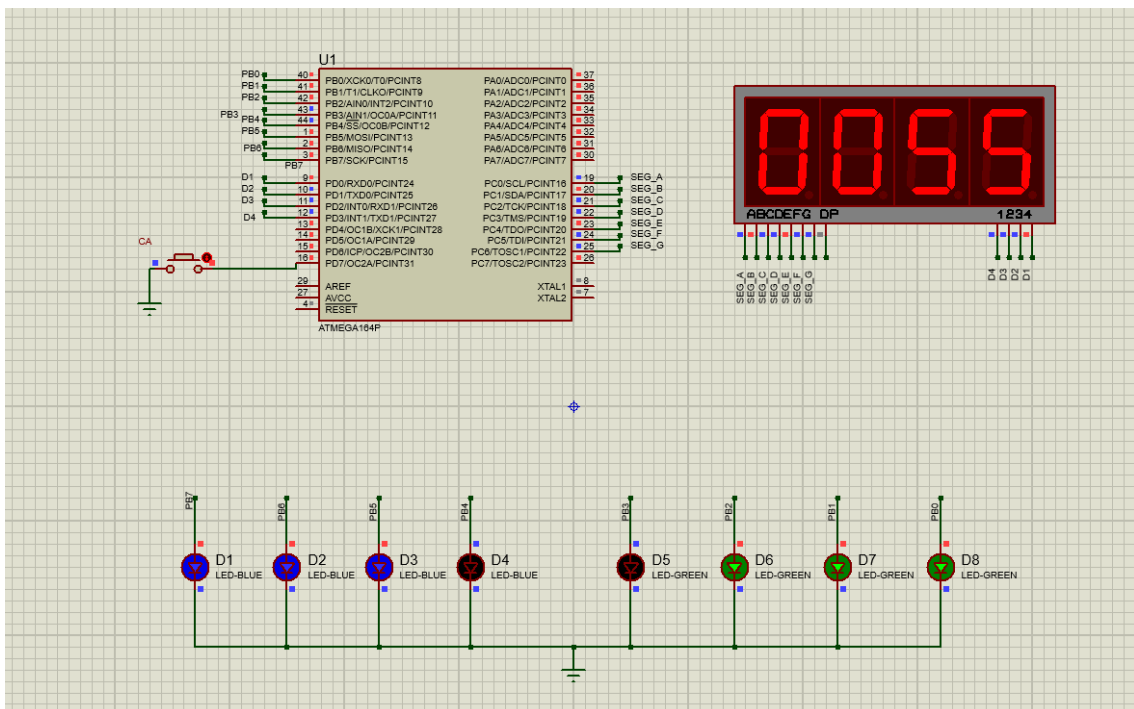


Figura 17 – Stare succesivă

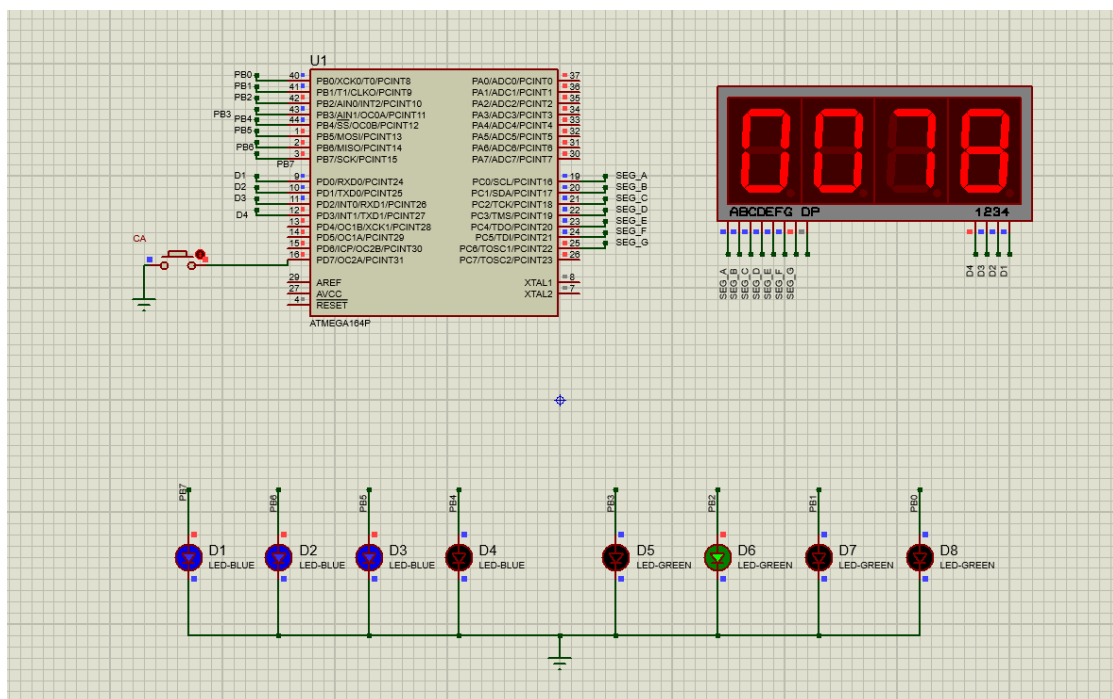


Figura 18 – Stare succesivă

De remarcat este faptul că la întoarcerea în starea inițială valoarea consumului este actualizată, semn că sistemul contorizează consumul de curent pe fundal.

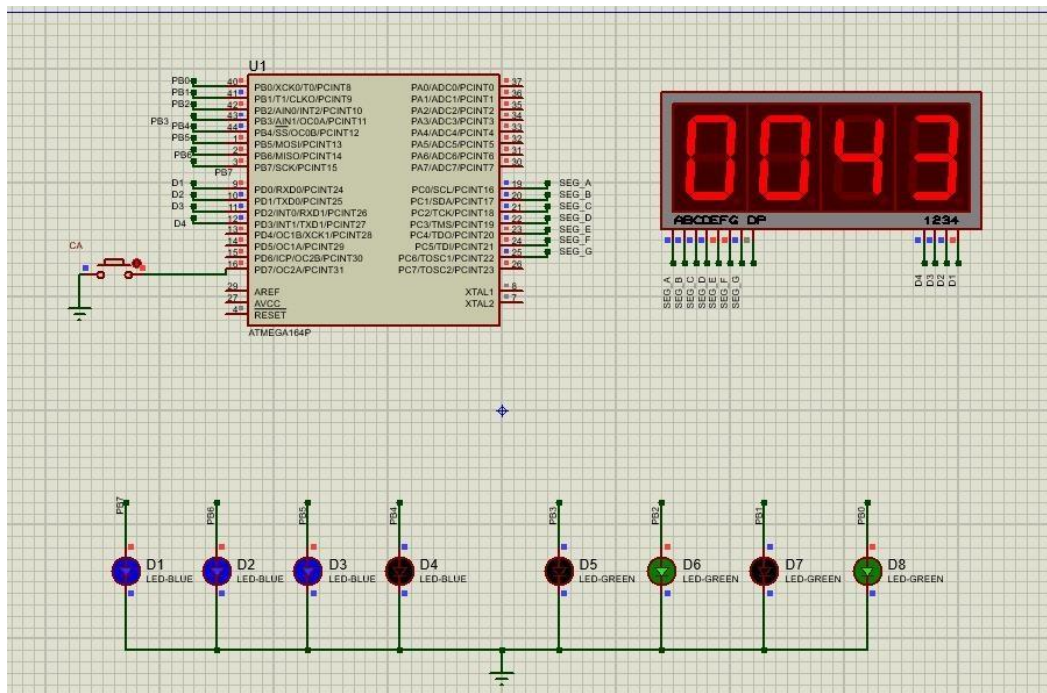


Figura 19 – Întoarcerea în starea inițială

- 2) Modul de contorizare fără intervale orare (setarea variabilei `MODE = 1`) este marcat prin aprinderea LED 4, respectiv stingerea LED 5 – 7. Funcționalitatea butonului/comenzii CA este inhibată în acest mode de lucru.

```

305 while (1)
306 {
307     // Display the consumption
308     DisplayConsumption();
309
310     // Wait for interruptions
311 }
312 }
313
314 // Function definitions
315 void Init()
316 {
317     // Setting initial states = 0
318     Q = Q1 = S1 = S2 = S3 = S_PULSE = 0;
319
320     // Setting the working mode
321     MODE = 1;
322 }
323
324 void UpdateConsumption()
325 {
326     // Identify PULSE
327     // PULSE = PINA & 0x01;
328
329     /* switch(S2)
330     {

```

Figura 20 – Setarea modului de lucru fără interval orare

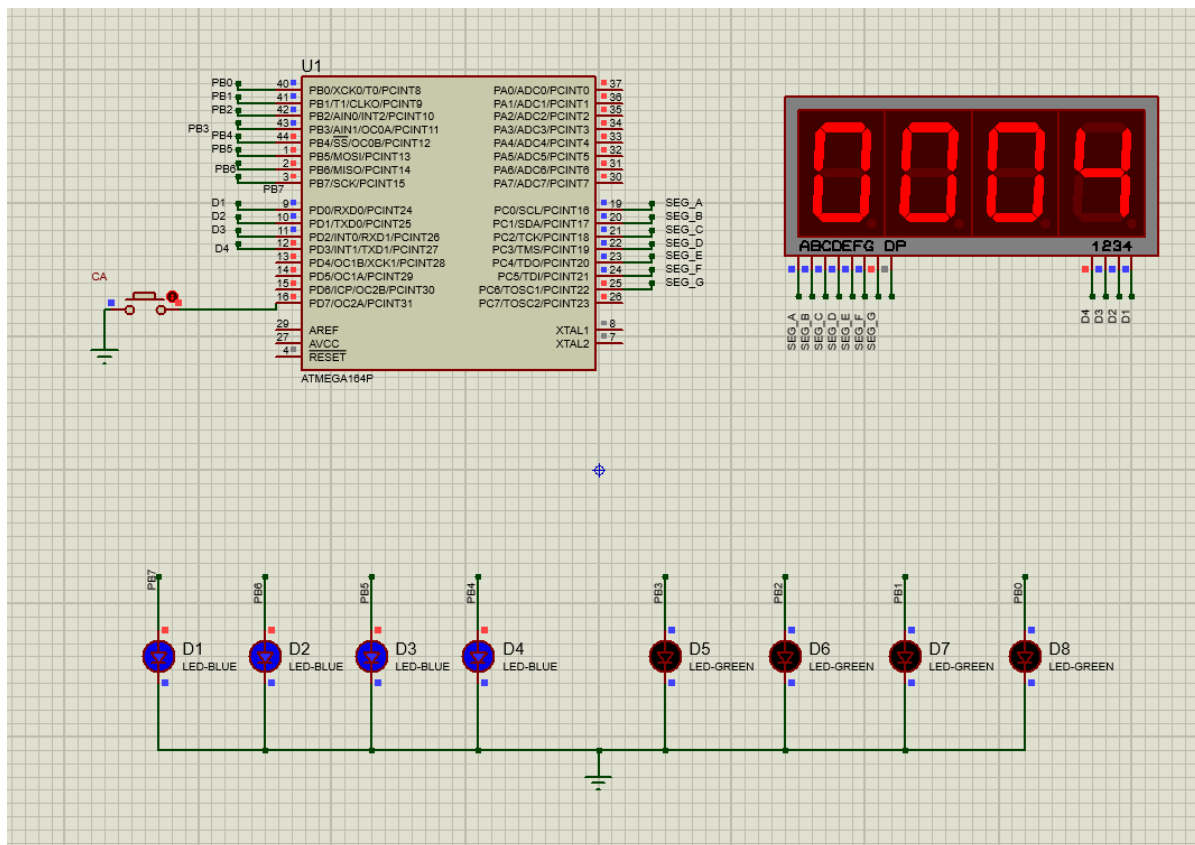


Figura 21 – Testarea modului de lucru fără intervale orare

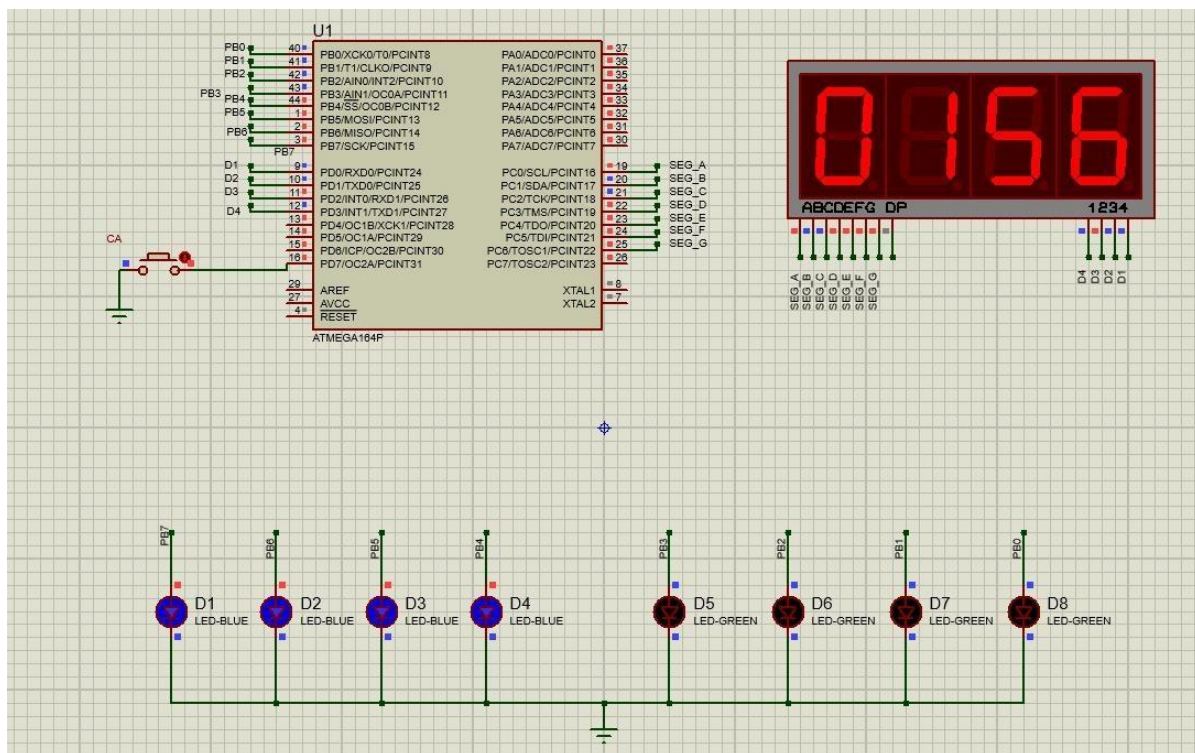


Figura 22 – Testarea modului de lucru fără intervale orare (2)



Se poate observa contorizarea continuă a consumului.

O altă informație afișată de ATmega164 pe cele 8 LED-uri este reprezentată de nivelul puterii consumate. Am ilustrat această funcționalitate printr-un proiect de test, în care am variat manual nivelul puterii consumate și am vizualizat pe rând comportamentul în Proteus 8 Professional. Se va observa cum valoarea variabilei PowerLevel este 0, LED-urile 1-3 vor fi stinse. Când PowerLevel este între valorile 0 și 3 kW se aprinde LED 3. Analog când PowerLevel este între valorile 3 și 6 kW se aprind LED-urile 2 și 3. Dacă PowerLevel trece de pragul de 6 kW (valoarea maximă fiind 10 kW), LED-urile 1-3 vor fi aprinse.

```
// For testing purposes, we will assume PowerLevel = 0 kW
PowerLevel = 0;
```

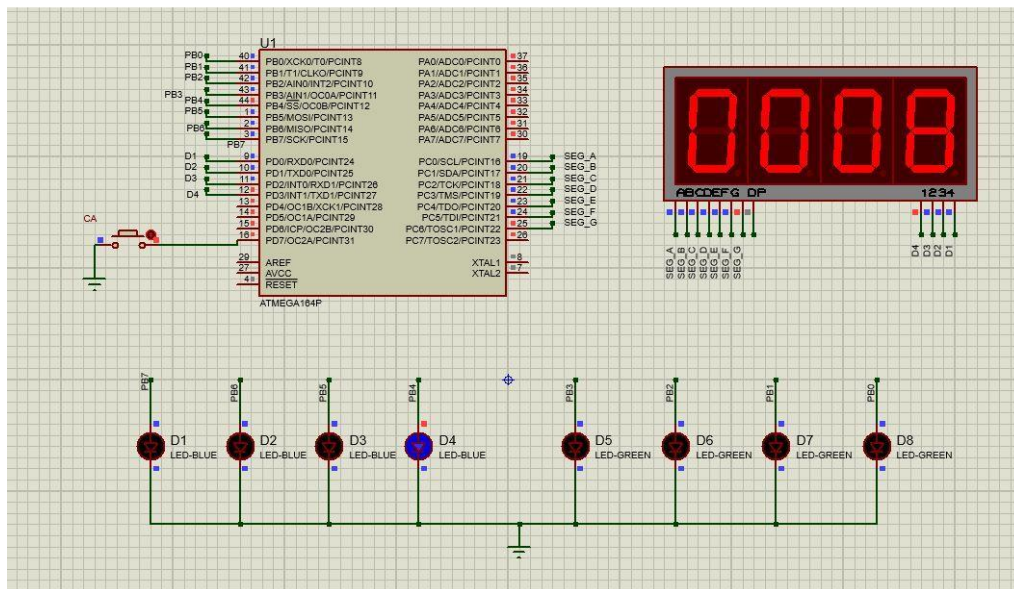


Figura 23 – Testarea unui nivel de putere PowerLevel = 0kW

```
// For testing purposes, we will assume PowerLevel = 2 kW
PowerLevel = 2;
```

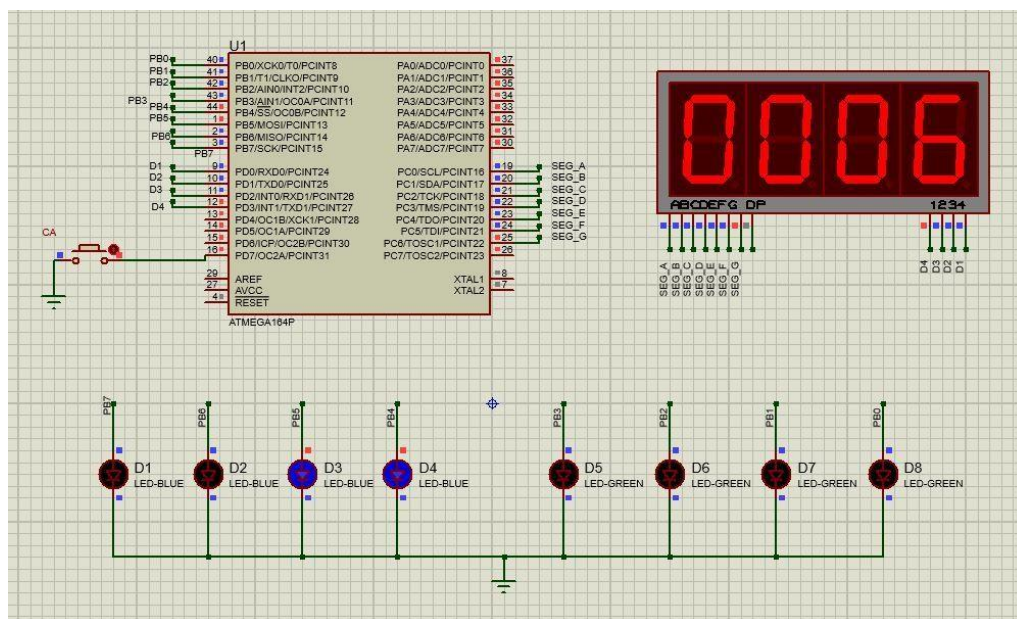


Figura 24 – Testarea unui nivel de putere PowerLevel < 3kW

```
// For testing purposes, we will assume PowerLevel = 5 kW
PowerLevel = 5;
```

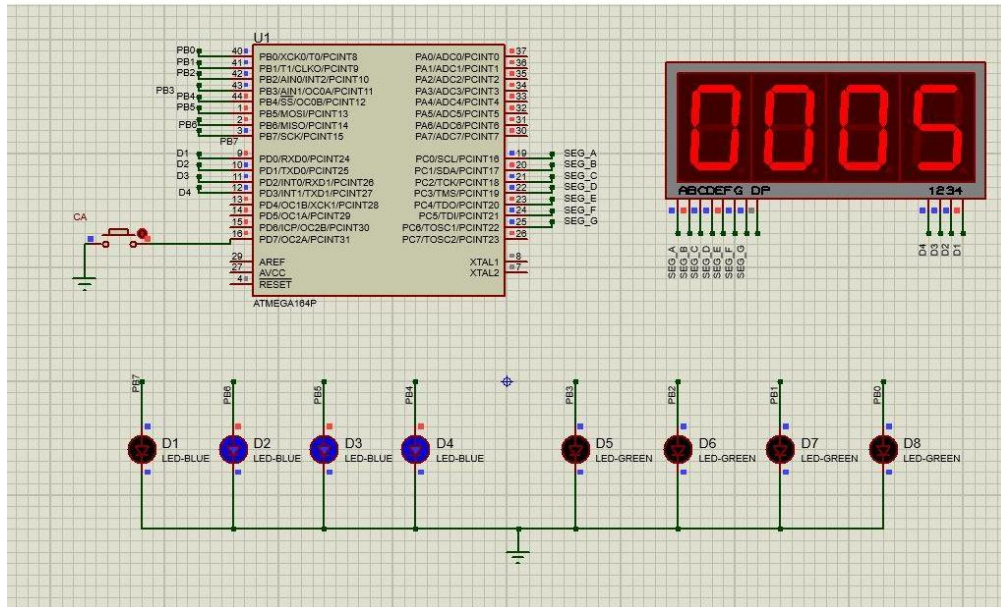


Figura 25 – Testarea unui nivel de putere PowerLevel < 6kW

```
// For testing purposes, we will assume PowerLevel = 10 kW
PowerLevel = 10;
```

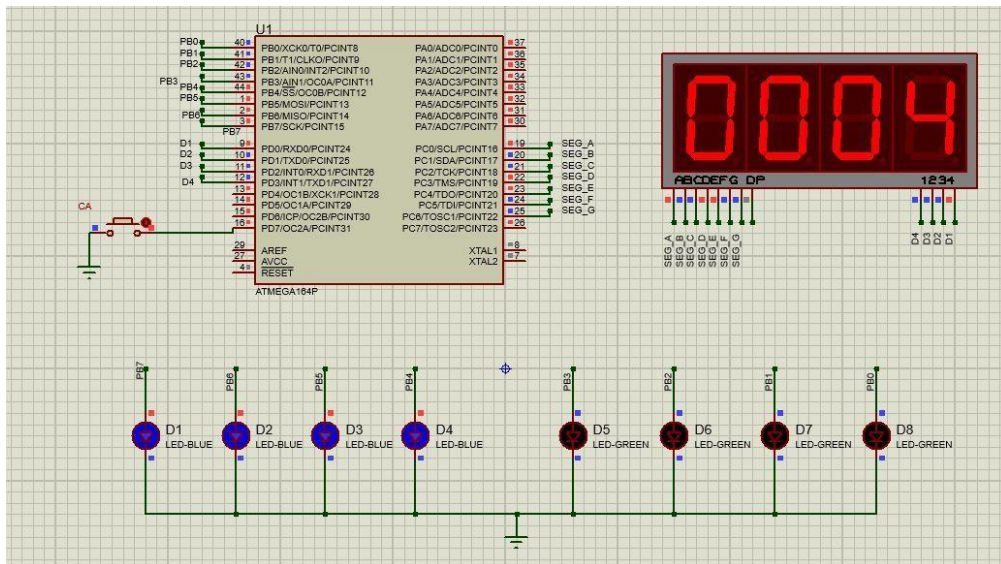


Figura 26 – Testarea unui nivel de putere PowerLevel = 10kW

## • ADSP 2181

Pentru testarea microcontroller-ului ADSP am realizat un proiect de test cu următoarele specificații:

- Am considerat succesiv următoarele valori de intrare (constante, setate manual de către programator):

U [V]	I [A]	P [W]
50	2	100
100	5	500
200	10	2000

- Numărul de pulsuri/kWh = 4.
- Intervalul de eșantionare  $dT \sim 1s$  (cu toate acestea, pentru a transmite pulsuri mai frecvent și pentru a putea vizualiza mai ușor comportamentul microcontrollerului am înmulțit puterea obținută cu o constantă mai mare).
- Am scris valoarea pulsului în fișierul de ieșire la fiecare unitate de timp de eșantionare  $dT$ . Prin urmare axa OX va reprezenta timpul în unități de timp ( $dT$ ), iar pe axa OY vom reprezenta valoarea pulsului la respectivul moment de timp.
- Comportamentul așteptat este ca numărul de pulsuri vizualizate pe reprezentarea de tip "plot" să fie direct proportional cu puterea determinată de cele două intrări.

Rezultatele experimentale obținute:

### 1. U = 50V, I = 2A, P = 100W:

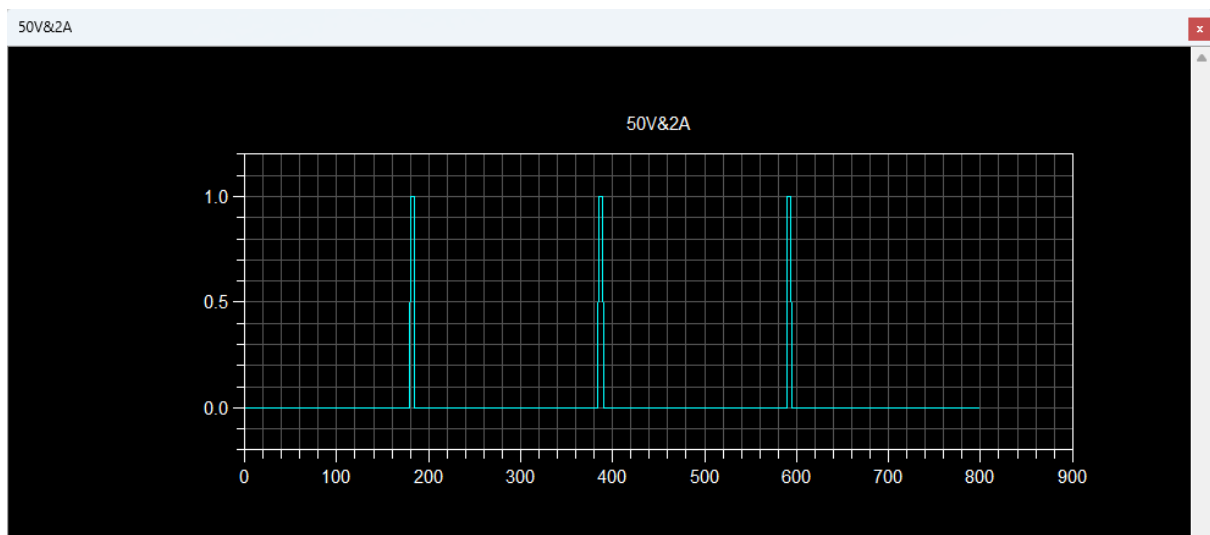


Figure 23 – PULSE =  $f(u.t.)$

2.  $U = 100V$ ,  $I = 5A$ ,  $P = 500W$

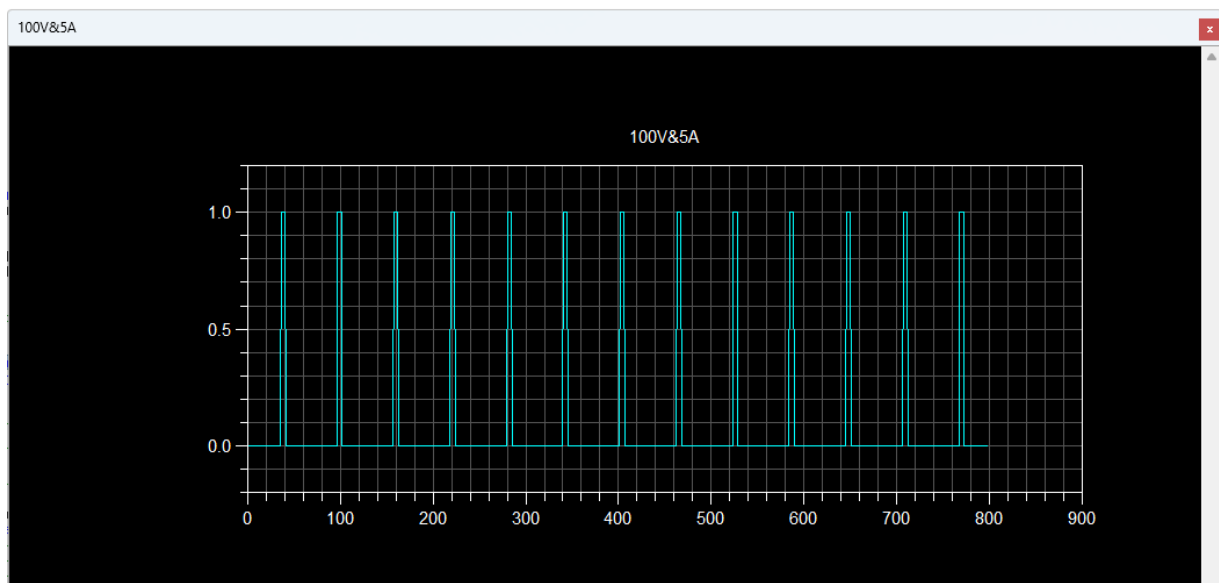


Figure 24 – PULSE =  $f(u.t.)$

3.  $U = 200V$ ,  $I = 10A$ ,  $P = 2kW$ :

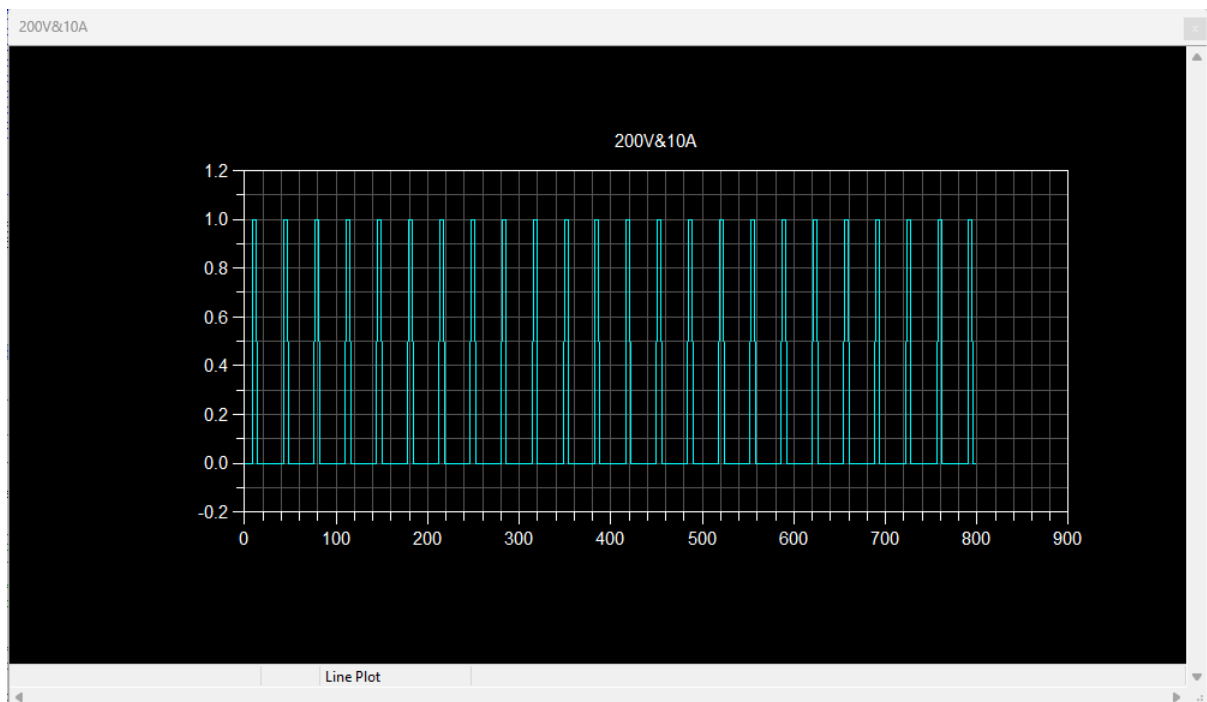


Figure 25 – PULSE =  $f(u.t.)$

## VI. COD SURSĂ

- **ATmega164:**

```

/*****
This program was created by the CodeWizardAVR V3.49a
Automatic Program Generator
© Copyright 1998-2022 Pavel Haiduc, HP InfoTech S.R.L.
http://www.hpinfotech.ro

Project :
Version :
Date    : 3/11/2023
Author  :
Company :
Comments:

Chip type           : ATmega164A
Program type        : Application
AVR Core Clock frequency: 10.000000 MHz
Memory model         : Small
External RAM size    : 0
Data Stack size      : 256
*****/

// I/O Registers definitions
#include <mega164a.h>
//#include <util/delay.h>

// Useful definitions
// Numar perioade necesare duratei unui
// puls intreg (100 ms)
#define T 5

// Numar de perioade necesare
// duratei unui puls pozitiv (cu durata 20 ms)
#define DP 1

/// CLS definitions ///
char S1 = 0; // CLS state
const long int H1 = 8;
const long int H2 = 16;

const long int Ter = 0x10000000;

long int A0[]={0x00000000+H1<<16, 1, 0x01000000+H1<<16, 1,
0x02000000+H1<<16, 1, 0x03000000+H1<<16, 1, 0x04000000+H1<<16, 1,
Ter , 0};
long int A1[]={0x00000000+H2<<16, 2, 0x01000000+H2<<16, 2,
0x02000000+H2<<16, 2, 0x03000000+H2<<16, 2, 0x04000000+H2<<16, 2,
Ter , 1};

```



```

long int A2[]={0x01000000      , 0, 0x02000000      , 0,
0x03000000      , 0, 0x04000000, 0, 0x05000000      , 3,
Ter , 2};
long int A3[]={0x00000000      , 0, Ter, 3};

char Tout[] = {0, 1, 2, 3};
long int *TAB_A[] = {A0, A1, A2, A3};
////////////////////////////////////

/// Time variables ///
char Z = 0; //day
char H = 0; //hour
char M = 0; //minutes
char S = 0; //seconds
////////////////////////////////////

char cnt_time = 0; //contor de timp
char T_SEC; // numar de perioade necesare pentru a acoperi 1 sec
char S2; //starea de contorizare a PS-ului

//////// Global variables //////////
// PULSE
char PULSE;

// Working mode ///
// 0 -> range on
// 1 -> range off
char MODE = 0;

// Flag (1 -> mode not set yet)
char modeFlag = 1;
////////////////////////////////////

// State variables ( Q -> consumption range,
// S1 -> consumption counting state,
// S3 -> display state)
char Q,Q1,S3;

/// Consumption array
//          0 - H1    H1 - H2    H2 - 0    Sat - Sun    Total
//          ^         ^         ^         ^         ^
int CONSUM[] = {0,      0,      0,      0,      0};

// Digits
char C4, C3, C2, C1;

// CA
char CA;

//Power Level
char PowerLevel = 0;

// Digit patterns (CLC)
const char DIGITS[] = {
    0b11000000, // 0

```

```

    0b11111001, // 1
    0b10100100, // 2
    0b10110000, // 3
    0b10011001, // 4
    0b10010010, // 5
    0b10000010, // 6
    0b11111000, // 7
    0b10000000, // 8
    0b10010000 // 9
};
////////////////////////////////////

// Power Level (CLC) ///
// char CLC_LEVEL[] = {0x00, 0x10, 0x30, 0x70, 0xF0}; // 4 levels
char CLC_LEVEL[] = {0x00, 0x20, 0x60, 0xE0}; // 3 levels
////////////////////////////////////

// Consumption range output (CLC) //
char CLC_RANGE_OUTPUT[] = {0x01, 0x02, 0x03, 0x00};
////////////////////////////////////

// Pulses contor
char cntP = 0;

///// Function headers /////
void Init();
void UpdateConsumption();
void DisplayConsumption();
void DisplayDigit(char currentDisplay, char digit);
void UpdateTime();
void CLS();
void DisplayPowerLevel();
void DisplayConsumptionDisplayMode();
void DisplayInfo();
////////////////////////////////////

//////// SCI //////////
// Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
    // Reinitialize Timer 0 value
    TCNT0=0x3C;

    // Update CA
    CA = (PORTD & 0x20) >> 5;

    //DisplayInfo
    DisplayInfo();

    // Check for pulses coming from ADSP
    UpdateConsumption();
}
////////////////////////////////////

```

```

void main(void)
{
// Declare your local variables here

// Crystal Oscillator division factor: 1
#pragma optsize-
CLKPR=(1<<CLKPCE);
CLKPR=(0<<CLKPCE) | (0<<CLKPS3) | (0<<CLKPS2) | (0<<CLKPS1) |
(0<<CLKPS0);
#ifdef _OPTIMIZE_SIZE_
#pragma optsize+
#endif

// Input/Output Ports initialization
// Port A initialization
// Function: Bit7=Out Bit6=In Bit5=In Bit4=In Bit3=In Bit2=In
Bit1=In Bit0=In
DDRA=(1<<DDA7) | (0<<DDA6) | (0<<DDA5) | (0<<DDA4) | (0<<DDA3) |
(0<<DDA2) | (0<<DDA1) | (0<<DDA0);
// State: Bit7=1 Bit6=P Bit5=P Bit4=P Bit3=P Bit2=P Bit1=P Bit0=P
PORTA=(1<<PORTA7) | (1<<PORTA6) | (1<<PORTA5) | (1<<PORTA4) |
(1<<PORTA3) | (1<<PORTA2) | (1<<PORTA1) | (1<<PORTA0);

// Port B initialization
// Function: Bit7=Out Bit6=Out Bit5=Out Bit4=Out Bit3=Out Bit2=Out
Bit1=Out Bit0=Out
DDRB=(1<<DDB7) | (1<<DDB6) | (1<<DDB5) | (1<<DDB4) | (1<<DDB3) |
(1<<DDB2) | (1<<DDB1) | (1<<DDB0);
// State: Bit7=1 Bit6=1 Bit5=1 Bit4=1 Bit3=1 Bit2=1 Bit1=1 Bit0=1
PORTB=(1<<PORTB7) | (1<<PORTB6) | (1<<PORTB5) | (1<<PORTB4) |
(1<<PORTB3) | (1<<PORTB2) | (1<<PORTB1) | (1<<PORTB0);

// Port C initialization
// Function: Bit7=Out Bit6=Out Bit5=Out Bit4=Out Bit3=Out Bit2=Out
Bit1=Out Bit0=Out
DDRC=(1<<DDC7) | (1<<DDC6) | (1<<DDC5) | (1<<DDC4) | (1<<DDC3) |
(1<<DDC2) | (1<<DDC1) | (1<<DDC0);
// State: Bit7=1 Bit6=1 Bit5=1 Bit4=1 Bit3=1 Bit2=1 Bit1=1 Bit0=1
PORTC=(1<<PORTC7) | (1<<PORTC6) | (1<<PORTC5) | (1<<PORTC4) |
(1<<PORTC3) | (1<<PORTC2) | (1<<PORTC1) | (1<<PORTC0);

// Port D initialization
// Function: Bit7=In Bit6=In Bit5=Out Bit4=Out Bit3=Out Bit2=Out
Bit1=Out Bit0=Out
DDRD=(0<<DDD7) | (0<<DDD6) | (1<<DDD5) | (1<<DDD4) | (1<<DDD3) |
(1<<DDD2) | (1<<DDD1) | (1<<DDD0);
// State: Bit7=T Bit6=T Bit5=1 Bit4=1 Bit3=1 Bit2=1 Bit1=1 Bit0=1
PORTD=(0<<PORTD7) | (0<<PORTD6) | (1<<PORTD5) | (1<<PORTD4) |
(1<<PORTD3) | (1<<PORTD2) | (1<<PORTD1) | (1<<PORTD0);

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: 9.766 kHz
// Mode: Normal top=0xFF
// OC0A output: Disconnected
// OC0B output: Disconnected

```

```

// Timer Period: 20.07 ms
TCCR0A=(0<<COM0A1) | (0<<COM0A0) | (0<<COM0B1) | (0<<COM0B0) |
(0<<WGM01) | (0<<WGM00);
TCCR0B=(0<<WGM02) | (1<<CS02) | (0<<CS01) | (1<<CS00);
TCNT0=0x3C;
OCR0A=0x00;
OCR0B=0x00;

// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: Timer1 Stopped
// Mode: Normal top=0xFFFF
// OC1A output: Disconnected
// OC1B output: Disconnected
// Noise Canceler: Off
// Input Capture on Falling Edge
// Timer1 Overflow Interrupt: Off
// Input Capture Interrupt: Off
// Compare A Match Interrupt: Off
// Compare B Match Interrupt: Off
TCCR1A=(0<<COM1A1) | (0<<COM1A0) | (0<<COM1B1) | (0<<COM1B0) |
(0<<WGM11) | (0<<WGM10);
TCCR1B=(0<<ICNC1) | (0<<ICES1) | (0<<WGM13) | (0<<WGM12) | (0<<CS12)
| (0<<CS11) | (0<<CS10);
TCNT1H=0x00;
TCNT1L=0x00;
ICR1H=0x00;
ICR1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// Timer/Counter 2 initialization
// Clock source: System Clock
// Clock value: Timer2 Stopped
// Mode: Normal top=0xFF
// OC2A output: Disconnected
// OC2B output: Disconnected
ASSR=(0<<EXCLK) | (0<<AS2);
TCCR2A=(0<<COM2A1) | (0<<COM2A0) | (0<<COM2B1) | (0<<COM2B0) |
(0<<WGM21) | (0<<WGM20);
TCCR2B=(0<<WGM22) | (0<<CS22) | (0<<CS21) | (0<<CS20);
TCNT2=0x00;
OCR2A=0x00;
OCR2B=0x00;

// Timer/Counter 0 Interrupt(s) initialization
TIMSK0=(0<<OCIE0B) | (0<<OCIE0A) | (1<<TOIE0);

// Timer/Counter 1 Interrupt(s) initialization
TIMSK1=(0<<ICIE1) | (0<<OCIE1B) | (0<<OCIE1A) | (0<<TOIE1);

// Timer/Counter 2 Interrupt(s) initialization
TIMSK2=(0<<OCIE2B) | (0<<OCIE2A) | (0<<TOIE2);

```

```

// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
// INT2: Off
// Interrupt on any change on pins PCINT0-7: Off
// Interrupt on any change on pins PCINT8-15: Off
// Interrupt on any change on pins PCINT16-23: Off
// Interrupt on any change on pins PCINT24-31: Off
EICRA=(0<<ISC21) | (0<<ISC20) | (0<<ISC11) | (0<<ISC10) | (0<<ISC01)
| (0<<ISC00);
EIMSK=(0<<INT2) | (0<<INT1) | (0<<INT0);
PCICR=(0<<PCIE3) | (0<<PCIE2) | (0<<PCIE1) | (0<<PCIE0);

// USART0 initialization
// USART0 disabled
UCSR0B=(0<<RXCIE0) | (0<<TXCIE0) | (0<<UDRIE0) | (0<<RXEN0) |
(0<<TXEN0) | (0<<UCSZ02) | (0<<RXB80) | (0<<TXB80);

// USART1 initialization
// USART1 disabled
UCSR1B=(0<<RXCIE1) | (0<<TXCIE1) | (0<<UDRIE1) | (0<<RXEN1) |
(0<<TXEN1) | (0<<UCSZ12) | (0<<RXB81) | (0<<TXB81);

// Analog Comparator initialization
// Analog Comparator: Off
// The Analog Comparator's positive input is
// connected to the AIN0 pin
// The Analog Comparator's negative input is
// connected to the AIN1 pin
ACSR=(1<<ACD) | (0<<ACBG) | (0<<ACO) | (0<<ACI) | (0<<ACIE) |
(0<<ACIC) | (0<<ACIS1) | (0<<ACIS0);
ADCSRB=(0<<ACME);
// Digital input buffer on AIN0: On
// Digital input buffer on AIN1: On
DIDR1=(0<<AIN0D) | (0<<AIN1D);

// ADC initialization
// ADC disabled
ADCSRA=(0<<ADEN) | (0<<ADSC) | (0<<ADATE) | (0<<ADIF) | (0<<ADIE) |
(0<<ADPS2) | (0<<ADPS1) | (0<<ADPS0);

// SPI initialization
// SPI disabled
SPCR=(0<<SPIE) | (0<<SPE) | (0<<DORD) | (0<<MSTR) | (0<<CPOL) |
(0<<CPHA) | (0<<SPR1) | (0<<SPR0);

// TWI initialization
// TWI disabled
TWCR=(0<<TWEA) | (0<<TWSTA) | (0<<TWSTO) | (0<<TWEN) | (0<<TWIE);

// Globally enable interrupts
#asm("sei")

// Initialize the device
Init();

```

```

while (1)
{
    // Display the consumption
    DisplayConsumption();

    // Wait for interruptions
}

///// Function definitions /////
void Init()
{
    // Setting initial states = 0
    Q = Q1 = S1 = S2 = S3 = 0;

    // Turn off displays
    PORTC = 0xff;
    PORTD = 0xff;
    PORTB = 0xff;
}

void UpdateConsumption()
{
    // Identify PULSE

    ///// PORT F //////////////////////////////////
    //       $\overline{M}$     $\overline{P}$   $\overline{O}$   $\overline{W}$     $\overline{E}$   $\overline{E}$   $\overline{R}$   $\overline{PU}$  //

    // Reading the power level
    PowerLevel = (PINA & 0x7E) >> 1;

    switch(S2)
    {
        case 0:
        {
            // If PULSE is on, start counting
            if (PULSE)
            {
                // Increment cntP
                cntP += 1;

                // Reset reading flag
                // PORTD &= 0x7f;

                if (modeFlag)
                {
                    MODE = (PINA & 0x80) >> 7;
                    modeFlag = 0;
                }

                // Go further if the pulse period has passed,
                // otherwise go back wait for sending ack again.
                S2 = (cntP == DP) ? 1 : 0;
            }
            break;
        }
    }
}

```

```

    }
    case 1:
    {
        if (~PULSE)
        {
            // Update current consumption range
            CLS();

            // Increment consumption
            if (MODE == 0)
            {
                CONSUM[Q] += 1;    // Working range on
            }
            else
            {
                CONSUM[4] += 1;    // Working range off
            }

            // Wait for another pulse
            S2 = 0;
            cntP = 0;
        }
        break;
    }
}

void DisplayConsumption()
{
    // We assume:
    // PORTC: PC0 - PC6 -> 7 segments (A-G)
    // PORTD: PD0 - PD3 -> select the common cathode for each digit
    (multiplexing)
        // PD3 - C4, PD2 - C3, PD1 - C2, PD0 - C1
    // Q - consumption range:
        // 0 -> 00:00 - H1:00
        // 1 -> H1:00 - H2:00          (MON - FRI)
        // 2 -> H2:00 - 00:00 (next day)
        // 3 -> SAT - SUN

    // The actual approach:
    // Each main loop iteration we multiplex the digits and display
    one at a time

    // If MODE = 1 -> display total consumption,
    // else -> display consumption based on current range.
    int cons = (MODE) ? CONSUM[4] : CONSUM[Q1];

    // Compute and display C4
    C4 = cons / 1000;
    cons %= 1000;
    DisplayDigit(4, C4);

    // Compute and display C3
    C3 = cons / 100;

```

```

    cons %= 100;
    DisplayDigit(3, C3);

    // Compute and display C2
    C2 = cons / 10;
    DisplayDigit(2, C2);

    // Compute and display C1
    C1 = cons % 10;
    DisplayDigit(1, C1);
}

void DisplayDigit(char currentDisplay, char digit)
{
    // Set PORTC pins to the corresponding digit
    // PORTC = DIGITS[digit];

    // Select the desired display (turn on the pin
    // corresponding to the desired digit (C4/C3/C2/C1)
    char output = 0xff;

    switch (currentDisplay)
    {
        case 4:
            // Turn PD3 on
            //output &= 0b00000111;
            output = 0x08;
            break;
        case 3:
            // Turn PD2 on
            // output &= 0b00001011;
            output = 0x04;
            break;
        case 2:
            // Turn PD1 on
            output = 0x02;
            break;
        case 1:
            // Turn PD0 on
            output = 0x01;
            break;
    }

    // Assign output to PORTC in order to select the desired
display;
    PORTD = output;

    // Set PORTC pins to the corresponding digit
    PORTC = DIGITS[digit];

    // Add delay (10 us)
    //_display_us(10);
}

void UpdateTime(){

```



```

    cnt_time += 1; //incrementare contor de timp
    if(cnt_time != T_SEC) return;

    cnt_time = 0; // se reseteaza contorul
    S+=1; //incrementeaza contor secunde

    if(S!=60) return;
    S = 0; //se reseteaza nr de secunde
    M += 1; //incrementeaza contor minute

    if(M!=60) return;
    M = 0;
    H += 1;

    if(H!=24) return;
    H = 0;
    Z += 1;

    if (Z == 7) Z = 0;
    return;
}

void CLS()
{
    //exemplu
    // Ziua 3, ora 8, min 6, sec 3
    //0x03080603
    long int now = (Z<<24) | (H<<16) | (M<<8) | S;

    long int *adr = TABA[Q];
    char ready = 0;
    int i = 0;
    long int out = 0;

    while (!ready)
    {
        if (now == adr[i]) {
            S1 = adr[i + 1];
            ready = 1; // Stop iterating through while
        }
        else if (adr[i] == T) ready = 1;
        else i = i+2;
    }
}

void DisplayInfo()
{
    DisplayConsumptionDisplayMode();
    DisplayPowerLevel();
}

void DisplayPowerLevel()
{
    char out;

```

```

    if (!PowerLevel)          // PowerLevel = 0 kW
    {
        out = CLC_LEVEL[0];
    }
    else if (PowerLevel < 3)   // 0 < PowerLevel < 3 kW
    {
        out = CLC_LEVEL[1];
    }
    else if (PowerLevel < 6)   // 3 <= PowerLevel < 6 kW
    {
        out = CLC_LEVEL[2];
    }
    else                       // PowerLevel >= 6 kW
    {
        out = CLC_LEVEL[3];
    }

    // Delete PB7-PB5
    PORTB &= 0x1f;

    // Display out on PB7-PB5
    PORTB |= out;
}

void DisplayConsumptionDisplayMode()
{
    char out;

    if (MODE == 1) // Working without ranges
    {
        // Clear PB4-0
        PORTB &= 0xE0;

        // Display on PB4-0
        PORTB |= 0x10;

        return;
    }

    switch(S3)
    {
        case 0:
        {
            if (CA == 0)          // Pressed CA
            {
                S3 = 1;
            }
            break;
        }
        case 1:                  // Released CA
        {
            if (CA)
            {
                S3 = 2;
                Q1 = 1;
            }
        }
    }
}

```

```

        break;
    }
    case 2:                // Pressed CA
    {
        if (CA == 0)
        {
            S3 = 3;
        }
        break;
    }
    case 3:                // Released CA
    {
        if (CA)
        {
            S3 = 4;
            Q1 = 2;
        }
        break;
    }
    case 4:
    {
        if (CA == 0)
        {
            S3 = 5;
        }
        break;
    }
    case 5:
    {
        if (CA)
        {
            S3 = 6;
            Q1 = 3;
        }
        break;
    }
    case 6:
    {
        if (CA == 0)
        {
            S3 = 7;
        }
        break;
    }
    case 7:
    {
        if (CA)
        {
            S3 = 0;
            Q1 = 0;
        }
        break;
    }
}

out = CLC_RANGE_OUTPUT[Q1] | (CLC_RANGE_OUTPUT[Q] << 2);

```

```

// Delete PB4-PB0
PORTB &= 0xE0;

// Display out on PB3-PB0
PORTB |= out;
}
////////////////////////////////////

```

## - ADSP 2181

```
#include "def2181.h"
```

```
// Output port
#define PORT_OUT 0xFF
```

```
// Input port
#define PORT_IN 0x1FF
```

```
// DP and T
#define DP 5
#define T 25
```

```
.SECTION/DM          buf_var1;
.var  rx_buf[3];      /* Status + L data + R data */
```

```
.SECTION/DM          buf_var2;
.var  tx_buf[3] = 0xc000, 0x0000, 0x0000; /* Cmd + L data + R data */
```

```
.SECTION/DM          buf_var3;
.var  init_cmds[13] = 0xc002, /*
```

Left input control reg  
b7-6: 0=left line 1

1=left aux 1  
2=left line 2  
3=left line 1 post-mixed loopback  
b5-4: res  
b3-0: left input gain x 1.5 dB  
\*/

```
0xc102, /*
```

Right input control reg  
b7-6: 0=right line 1  
1=right aux 1  
2=right line 2  
3=right line 1 post-mixed loopback  
b5-4: res  
b3-0: right input gain x 1.5 dB

```
*/
```

```

0xc288, /*
    left aux 1 control reg
    b7 : 1=left aux 1 mute
    b6-5: res
    b4-0: gain/atten x 1.5, 08= 0dB, 00= 12dB
*/
0xc388, /*
    right aux 1 control reg
    b7 : 1=right aux 1 mute
    b6-5: res
    b4-0: gain/atten x 1.5, 08= 0dB, 00= 12dB
*/
0xc488, /*
    left aux 2 control reg
    b7 : 1=left aux 2 mute
    b6-5: res
    b4-0: gain/atten x 1.5, 08= 0dB, 00= 12dB
*/
0xc588, /*
    right aux 2 control reg
    b7 : 1=right aux 2 mute
    b6-5: res
    b4-0: gain/atten x 1.5, 08= 0dB, 00= 12dB
*/
0xc680, /*
    left DAC control reg
    b7 : 1=left DAC mute
    b6 : res
    b5-0: attenuation x 1.5 dB
*/
0xc780, /*
    right DAC control reg
    b7 : 1=right DAC mute
    b6 : res
    b5-0: attenuation x 1.5 dB
*/
0xc85c, /*
    data format register
    b7 : res
    b5-6: 0=8-bit unsigned linear PCM
    1=8-bit u-law companded
    2=16-bit signed linear PCM
    3=8-bit A-law companded
    b4 : 0=mono, 1=stereo
    b0-3: 0= 8.
    1= 5.5125
    2= 16.
    3= 11.025
    4= 27.42857
    5= 18.9

```

```

6= 32.
7= 22.05
8= .
9= 37.8
a= .
b= 44.1
c= 48.
d= 33.075
e= 9.6
f= 6.615
(b0) : 0=XTAL1 24.576 MHz; 1=XTAL2 16.9344 MHz
*/
0xc909, /*
        interface configuration reg
        b7-4: res
        b3 : 1=autocalibrate
        b2-1: res
        b0 : 1=playback enabled
*/
0xca00, /*
        pin control reg
        b7 : logic state of pin XCTL1
        b6 : logic state of pin XCTL0
        b5 : master - 1=tri-state CLKOUT
        slave - x=tri-state CLKOUT
        b4-0: res
*/
0xcc40, /*
THIS PROGRAM USES 16 SLOTS PER FRAME
        miscellaneous information reg
        b7 : 1=16 slots per frame, 0=32 slots per frame
        b6 : 1=2-wire system, 0=1-wire system
        b5-0: res
*/
0xcd00; /*
        digital mix control reg
        b7-2: attenuation x 1.5 dB
        b1 : res
        b0 : 1=digital mix enabled
*/

.SECTION/DM          data1;
.var    stat_flag;

// PF port
.var    PF_input;
.var    PF_output;

.var cntP = 0;        // period counter
.var dT;              // Sampling period

```

```

.var MODE = 0;           // Working mode (0 -> no working ranges)
.var U;                 // Voltage
.var I;                 // Current
.var E[2] = {0, 0};     // Energy (kWs)
.var n;                 // Number of pulses to send
.var Q;                 // PS state
.var cntG;              // Pulse generator counter
.var P;                 // Power
.var Threshold[2] = {0, 0}; // Power threshold to generate a pulse
.var PulsesNumber;      // Pulses number / KWh
.var PulsesNumberIndex;
.var dTIndex;
.var noInterrupts = 5; // No of interrupts to cover 20ms
.var cntInterrupts = 0; // Interrupts counter

.var TAB_PULSES[4] = {1, 2, 4, 8}; // Number of pulses/kWh -> 1/2/4/8;
.var TAB_SAMPLING_PERIODS[4] = {50, 3000, 30000, 60000}; // 1s/1m/5m/10m
.SECTION/PM              pm_da;

/**** Interrupt Vector Table ****/
.SECTION/PM  interrupts;
    jump start; rti; rti; rti; /*00: reset */

    jump sci;   rti; rti; rti; /*04: IRQ2 */

    rti;        rti; rti; rti; /*08: IRQL1 */
    rti;        rti; rti; rti; /*0c: IRQL0 */
    ar = dm(stat_flag); /*10: SPORT0 tx */
    ar = pass ar;
    if eq rti;
    jump next_cmd;
    jump sci; /*14: SPORT0 rx */
    rti; rti; rti;
    rti;        rti; rti; rti; /*18: IRQE */
    rti;        rti; rti; rti; /*1c: BDMA */
    rti;        rti; rti; rti; /*20: SPORT1 tx or IRQ1 */
    rti;        rti; rti; rti; /*24: SPORT1 rx or IRQ0 */
    nop; rti; rti; rti; /*28: timer */
    rti;        rti; rti; rti; /*2c: power down */

.SECTION/PM              seg_code;
/*****
*****
*
* ADSP 2181 intialization
*

```

```

*****
****/

```

```

start:

```

```

    /* shut down sport 0 */
    ax0 = b#0000100000000000;
        dm (Sys_Ctrl_Reg) = ax0;
    ena timer;

```

```

    i5 = rx_buf;
    l5 = LENGTH(rx_buf);
    i6 = tx_buf;
    l6 = LENGTH(tx_buf);
    i3 = init_cmds;
    l3 = LENGTH(init_cmds);

```

```

    m1 = 1;
    m5 = 1;

```

```

/*===== SERIAL PORT #0 STUFF
=====*/

```

```

    ax0 = b#0000110011010111; dm (Sport0_Autobuf_Ctrl) = ax0;
    /* |||!|-/!|-/|/+ receive autobuffering 0=off, 1=on
       |||! ! | | +-- transmit autobuffering 0=off, 1=on
       |||! ! | +---- | receive m?
       |||! ! |      | m5
       |||! ! +----- ! receive i?
       |||! !          ! i5
       |||! !          !
       |||! +===== | transmit m?
       |||!          | m5
       |||!+----- ! transmit i?
       |||!          ! i6
       |||!          !
       |||+===== | BIASRND MAC biased rounding control bit
       ||+-----0
       |+----- | CLKODIS CLKOUT disable control bit
       +-----0
    */

```

```

    ax0 = 0; dm (Sport0_Rfsdiv) = ax0;
    /* RFSDIV = SCLK Hz/RFS Hz - 1 */
    ax0 = 0; dm (Sport0_Sclkdiv) = ax0;
    /* SCLK = CLKOUT / (2 (SCLKDIV + 1) */
    ax0 = b#1000011000001111; dm (Sport0_Ctrl_Reg) = ax0;
    /* multichannel
       ||+--/!||+/-/ | number of bit per word - 1
       ||| !|||      | = 15
       ||| !|||      |

```



```

||| !||| |
||| !|||+===== ! 0=right just, 0-fill; 1=right just, signed
||| !||| ! 2=compond u-law; 3=compond A-law
||| !||+ -----receive framing logic 0=pos, 1=neg
||| !||+ ----- transmit data valid logic 0=pos, 1=neg
||| |+===== RFS 0=ext, 1=int
||| + ----- multichannel length 0=24, 1=32 words
||+ -----| frame sync to occur this number of clock
|| | cycle before first bit
|| |
|| |
||+ ----- ISCLK 0=ext, 1=int
+----- multichannel 0=disable, 1=enable
*/
/* non-multichannel
|||!|||!|||!+---/ | number of bit per word - 1
|||!|||!|||! | = 15
|||!|||!|||! |
|||!|||!|||! |
|||!|||!|||!+===== ! 0=right just, 0-fill; 1=right just, signed
|||!|||!|||!+----- ! 2=compond u-law; 3=compond A-law
|||!|||!|||+ -----receive framing logic 0=pos, 1=neg
|||!|||!|||+ ----- transmit framing logic 0=pos, 1=neg
|||!|||!|||+===== RFS 0=ext, 1=int
|||!|||+ ----- TFS 0=ext, 1=int
|||!||+ -----TFS width 0=FS before data, 1=FS in sync
|||!||+----- TFS 0=no, 1=required
|||+===== RFS width 0=FS before data, 1=FS in sync
||+ -----RFS 0=no, 1=required
||+ ----- ISCLK 0=ext, 1=int
+----- multichannel 0=disable, 1=enable
*/

```

```

ax0 = b#0000000000000000111; dm (Sport0_Tx_Words0) = ax0;
/* ^15 00^ transmit word enables: channel # == bit # */
ax0 = b#0000000000000000111; dm (Sport0_Tx_Words1) = ax0;
/* ^31 16^ transmit word enables: channel # == bit # */
ax0 = b#0000000000000000111; dm (Sport0_Rx_Words0) = ax0;
/* ^15 00^ receive word enables: channel # == bit # */
ax0 = b#0000000000000000111; dm (Sport0_Rx_Words1) = ax0;
/* ^31 16^ receive word enables: channel # == bit # */

```

```

/*===== SYSTEM AND MEMORY STUFF
=====*/
ax0 = b#000110000000000000; dm (Sys_Ctrl_Reg) = ax0;
/* +-/!||+ ---- /+/- | program memory wait states
| !||| | 0
| !||| |

```

```

| !||+-----0
| !||      0
| !||      0
| !||      0
| !||      0
| !||      0
| !||      0
| !||      0
| !|+----- SPORT1 1=serial port, 0=FI, FO, IRQ0, IRQ1,..
| !+-----SPORT1 1=enabled, 0=disabled
| +===== SPORT0 1=enabled, 0=disabled
+-----0
      0
      0
*/

```

```

ifc = b#00000011111110;    /* clear pending interrupt */
nop;

```

```

icntl = b#00010;
/*  |||+ - | IRQ0: 0=level, 1=edge
   |||+ - - | IRQ1: 0=level, 1=edge
   ||+ - - - | IRQ2: 0=level, 1=edge
   |+ - - - 0
   | - - - - | IRQ nesting: 0=disabled, 1=enabled
*/

```

```

mstat = b#1100000;
/*  |||||+ - | Data register bank select
   |||||+ - - | FFT bit reverse mode (DAG1)
   ||||+ - - - | ALU overflow latch mode, 1=sticky
   |||+ - - - - | AR saturation mode, 1=saturate, 0=wrap
   ||+ - - - - - | MAC result, 0=fractional, 1=integer
   |+ - - - - - | timer enable
   + - - - - - | GO MODE
*/

```

```

jump skip;

```

```

/*****
*****
*
* ADSP 1847 Codec initialization
*
*****
*****/

```

```

/* clear flag */
ax0 = 1;
dm(stat_flag) = ax0;

/* enable transmit interrupt */
ena ints;
imask = b#0001000001;
/*  |||||+ | timer
   |||||+- | SPORT1 rec or IRQ0
   |||||+-- | SPORT1 trx or IRQ1
   |||||+--- | BDMA
   ||||+---- | IRQE
   |||+----- | SPORT0 rec
   ||+----- | SPORT0 trx
   |+----- | IRQL0
   |+----- | IRQL1
   +----- | IRQ2
*/

ax0 = dm (i6, m5);      /* start interrupt */
tx0 = ax0;

check_init:
ax0 = dm (stat_flag);    /* wait for entire init */
af = pass ax0;           /* buffer to be sent to */
if ne jump check_init;    /* the codec */

ay0 = 2;
check_aci1:
ax0 = dm (rx_buf);       /* once initialized, wait for codec */
ar = ax0 and ay0;        /* to come out of autocalibration */
if eq jump check_aci1;    /* wait for bit set */

check_aci2:
ax0 = dm (rx_buf);       /* wait for bit clear */
ar = ax0 and ay0;
if ne jump check_aci2;
idle;

ay0 = 0xbf3f;            /* unmute left DAC */
ax0 = dm (init_cmds + 6);
ar = ax0 AND ay0;
dm (tx_buf) = ar;
idle;

ax0 = dm (init_cmds + 7); /* unmute right DAC */
ar = ax0 AND ay0;
dm (tx_buf) = ar;

```

```

idle;

ifc = b#00000011111110;  /* clear any pending interrupt */
nop;

        imask = b#0001110001;  /* enable rx0 interrupt */
/*  |||||+ | timer
    |||||+- | SPORT1 rec or IRQ0
    |||||+-- | SPORT1 trx or IRQ1
    |||||+--- | BDMA
    ||||+---- | IRQE
    |||+----- | SPORT0 rec
    ||+----- | SPORT0 trx
    |+----- | IRQL0
    |+----- | IRQL1
    +----- | IRQ2
*/

/* end codec initialization, begin filter demo initialization */

skip: imask = 0x200;

// wait states

si=0xFFFF;
dm(Dm_Wait_Reg)=si;

// call init;

// PF ports
si=0x00ff;
dm(Prog_Flag_Comp_Sel_Ctrl)=si; // PF0-7 outputs

ena m_mode;
ax0 = 0;
dm(MODE) = ax0;                // Mode 0 => working range on
ax0 = 50;
dm(dT) = ax0;                  // Interr -> every 20ms => dT = 50
ax0 = 4;
dm(PulsesNumber) = ax0;        // Pulses / kWh

// COMPUTE_THRESHOLD:
ena m_mode;
ax1 = dm(PulsesNumber);        // Pulses/kWh
// af = PASS 0;

ay0 = 3600;                    // ay0 = 1kWh = 3600 kWhs
ay1 = 0;

```

```

DIVS ay1, ax1;
DIVQ ax1; DIVQ ax1;
DIVQ ax1; DIVQ ax1;
DIVQ ax1; DIVQ ax1;
DIVQ ax1; DIVQ ax1;
DIVQ ax1; DIVQ ax1;
DIVQ ax1; DIVQ ax1;
DIVQ ax1; DIVQ ax1;
DIVQ ax1; DIVQ ax1;

mx0 = ay0;
my0 = 1000;           // Get the result in Ws
mr = mx0 * my0 (uu);

dm(Threshold) = mr0;
dm(Threshold + 1) = mr1;

wt:

        nop;
        jump wt;

/* .....
-
- SPORT0 interrupt handler
-
..... */

sci:
        ena sec_reg

        ay0 = dm(Q);           // Read current state
        ar = PASS ay0;         // ar = 0 + Q
        if eq jump Q0;         // If ar = Q = 0 => jump towards Q0

        ar = ay0 - 1;          // ar = Q - 1
        if eq jump Q1;         // If ar = 0 => jump towards Q1

        ax0 = 2;
        ar = ay0 - ax0;         // ar = Q - 2
        if eq jump Q2;         // If ar = 0 => jump towards Q2

        ax0 = 3;
        ar = ay0 - ax0;         // ar = Q - 3
        if eq jump Q3;         // If ar = 0 => jump towards Q3

        ax0 = 4;
        ar = ay0 - ax0;         // ar = Q - 4
        if eq jump Q4;         // If ar = 0 => jump towards Q4

```

```

ax0 = 5;
ar = ay0 - ax0;          // ar = Q - 5
if eq jump Q5;           // If ar = 0 => jump towards Q5
rti;

```

Q0:

```

// Implementing a counter in order to sync with ATmega
// (which has timing interrupts at every 20ms;
ay0 = dm(cntInterrupts); // Read current cycles counter
ar = ay0 + 1;             // Increment the cycles counter
dm(cntInterrupts) = ar;
ax0 = ar;
ay0 = dm(noInterrupts);  // Get the necessary cycles number
af = ax0 - ay0;           // Compute cntInterrupts -
noInterruptsif eq jump GO_Q0; // If not equal => return
    ax1 = 1;
dm(Q) = ax1;
rti;

```

```

GO_Q0:
ax1 = 0;
dm(Q) = ax1;
rti;

```

////////////////////////////////////

//////////////////// Q = 1 //////////////////////

Q1:

```

// Check if the sampling period is complete
ay1 = dm(cntP);          // ay1 = cntP
ar = ay1 + 1;            // ar = cntP + 1
dm(cntP) = ar;           // cntP = ar = cntP + 1
ax1 = dm(dT);            // ax1 = dT
ay1 = dm(cntP);          // Refresh: ay1 = cntP (incremented)
ar = ax1 - ay1;          // ar = dT - cntP
if gt rti;               // if dT > cntP => return
ax1 = 0;
dm(cntP) = ax1;          // Restart counting

ax1 = 0;                  // Write PULSE = 0 at 0xFF
IO(PORT_OUT) = ax1;

```

```

// Read U & I
mx0 = dm(rx_buf + 2); // Citeste senzorii de tensiune & curent
my0 = dm(rx_buf + 1);

```

```

mx0 = 200;               // U
my0 = 10;                // I

```

```

// Compute dE

```

```

mr = mx0 * my0 (uu);    // mr = U * I
dm(P) = mr0;             // P (power) = U * I
mx1 = dm(dT);           // mx1 = dT
my1 = mr0;              // my1 = U * I
mr0 = dm(E);            // mr = E
mr1 = dm(E + 1);
mr = mr + mx1 * my1 (uu); // mr = E + U * I * dT = E + dE
dm(E) = mr0;            // Save E = E'
dm(E + 1) = mr1;

```

```

ax0 = dm(E);
ax1 = dm(E + 1);

```

```

ay0 = dm(Threshold);
ay1 = dm(Threshold + 1);

```

```

DIS AR_SAT;
    ar = ax0 - ay0;
    ar = ax1 - ay1 + C - 1, ax0 = ar;
    ax1 = ar;

```

```

if lt rti;

```

```

// COMPUTE_N:
si = 1;
DIVIDE:
// ax1 ax0 -> E
// ay1 ay0 -> TH
// Save last iteration's result:
dm(E) = ax0;
dm(E + 1) = ax1;

```

```

DIS AR_SAT;
    ar = ax0 - ay0;
    ar = ax1 - ay1 + C - 1, ax0 = ar;
    ax1 = ar;

```

```

if lt jump STOP;

```

```

ar = si;
ar = ar + 1;
si = ar;
jump DIVIDE;

```

```

STOP:
dm(n) = si;
ax1 = 2;
dm(Q) = ax1;           // Q = 2;
rti;

```

```

////////////////////////////////////

```

////////// Q = 2 //////////

Q2:

```
ax0 = 1;
ax1 = 3;
ay0 = 0;
    ay1 = dm(n);           // ay0 = n
af = PASS ay1;           // ar = n
if le ar = ax0 + ay0;    // if n <= 0 => return to Q0
if gt ar = ax1 + ay0;    // else => return to Q1
dm(Q) = ar;              // Go to Q = 2
rti;
```

//////////

////////// Q = 3 //////////

Q3:

```
// Generating the pulse //
ax1 = 1;
// dm(Prog_Flag_Data) = ax1; // PF = PPPP PPP1 (PULSE = 1)
IO(PORT_OUT) = ax1;      // Write PULSE = 1 at 0xFF
ay1 = dm(cntG);
ar = ay1 + 1;           // Increment cnt
dm(cntG) = ar;
ax1 = ar;               // ax1 = cnt
ay1 = DP;               // ay1 = DP
af = ax1 - ay1;
ar = 3;                 // Next state => default 3
if eq ar = ar + 1; // If cntG = DP => Go to Q4
dm(Q) = ar;             // Set Q state value
rti;
```

//////////

////////// Q = 4 //////////

Q4:

```
ax1 = 0;                // Write PULSE = 0 at 0xFF
IO(PORT_OUT) = ax1;
ay1 = dm(cntG);
ar = ay1 + 1;           // Increment cnt
dm(cntG) = ar;
ax1 = T;
ay1 = ar;
af = ax1 - ay1;         // Check whether cnt = T
ar = 4;
if eq ar = ar + 1;      // If so, go to Q = 4
dm(Q) = ar;
rti;
```

//////////



```

////////// Q = 5 //////////
Q5:
    ax0 = 0;
    dm(cntG) = ax0;
    ay1 = dm(n);
    ar = ay1 - 1;
    if gt jump GO_TO_Q3;
    // if le jump UPDATE_INFO;
    ax0 = 0;
    dm(Q) = ax0;
    rti;

GO_TO_Q3:
    ax0 = 3;
    dm(Q) = ax0;    // Switch to Q = 3
    rti;
//////////

/* .....
-
- transmit interrupt used for Codec initialization
-
..... */
next_cmd:
    ena sec_reg;
    ax0 = dm(i3, m1);    /* fetch next control word and */
    dm(tx_buf) = ax0;    /* place in transmit slot 0 */
    ax0 = i3;
    ay0 = init_cmds;
    ar = ax0 - ay0;
    if gt rti;           /* rti if more control words still waiting */
    ax0 = 0xaf00;        /* else set done flag and */
    dm(tx_buf) = ax0;    /* remove MCE if done initialization */
    ax0 = 0;
    dm(stat_flag) = ax0; /* reset status flag */
    rti;

```