Microprocesoare, Microcontrolere –Prelegere 5, partea I-a

1. Temporizatoare şi numărătoare (Timerer and counters). Generalităţi.

Marea majoritate a aplicaţiilor pentru microcontrolere necesită măsurarea tipului: cât timp încălzim mâncarea la cuptorul cu microunde, cât tip spală rufele maşina de spălat, etc. Până în prezent se pot imagina două modalități de măsurare a timpului. Ambele modalități se bazează pe acelaşi principiu: executarea de către procesor a unei instrucțiuni cod maşină necesită un timp bine precizat. În consecință, dacă procesorul a executat *n* instrucţiuni timpul scurs se află prin însumarea timpilor de execuţie a celor *n* instrucțiuni.

Prima modalitate de a măsura trecerea timpului a fost folosită în laboratorul blink. Programul blink este reluat în continuare:

```
#define P  125000L
#define DF 50L
#define TH (P*DF/100)
//125L = 1ms

int main(){
   volatile long i;
   DDRA=0xff;
   i=0;

   while(1){
      if(i==0)
         PORTA=1;  //aprinde LED-ul

      if(i==TH)
         PORTA=0;  //stinge LED-ul

      i++;
      if(i==P)
         i=0; //a trecut o secunda
   }
}
```

În acest program trecerea timpului este măsurată prin intermediul numărului de execuţii ale buclei principale `while(1)`. Din păcate, chiar şi din acest exemplu simplu, se observă că timpul de execuţie al buclei nu este întotdeauna acelaşi. Orice execuţie a buclei înseamnă executarea instrucţiunilor scrise cu negru bold. Există însă o buclă în care se execută suplimentar instrucţiunea care aprinde LED-ul, o a doua bucla în care se stinge LED-ul şi o a treia buclă în care se face `i=0`. Aceste trei bucle for avea un timp de execuţie diferit de timpul de execuţia a celorlalte 124997 de bucle. În cazul clipirii LED-ului această diferenţă între timpii de execuţie este imposibil de observat şi poate fi acceptată.

În anumite cazuri însă situația este mult mai proastă. În continuare este prezentată structură probabilă a buclei principale:

```
int main(){
   while(1){
      //citește intrările i1,i2,…ik. Consumă Tin
       if(i1 ……)     //Consumă Tv1
          //calcul 1; Consuma Tc1

       if(i2 ……)     //Consumă Tv2
          //calcul 2; Consuma Tc2

       //……
       if(ik ……)     //Consumă Tvk
```

```
        //calcul k; Consuma Tck

      //actualizează ieșirile. Consumă Tout
   }//end while 1
}
```

Mai întâi programul citeşte intrările i_1, i_2,.., i_k. Citirea intrărilor necesită timpul $\mathbf{T_{in}}$. Apoi fiecare intrare este testată pentru a vedea dacă este îndeplinită o condiție specifică. Executarea unui test consumă timpul $\mathbf{Tv_i}$ (timp de verificare pentru intrarea *i*). În funcție de rezultatul testului se execută sau nu anumite calcule. Timpul consumat pentru executarea calculului *i* se notează $\mathbf{Tc_i}$. În final se actualizează ieşirile, operaţie care consumă timpul **Tout**. Dacă toate condițiile din instrucţiunile *if* sunt evaluate la fals, timpul de execuţie este cel mai mic cu putinţă şi are valoarea:

$$T_{min}=T_{in}+Tv_1+Tv_2+\ldots+Tv_k+T_{out}$$

În schimb, dacă toate condiţiile din instrucţiunile *if* sunt evaluate la adevărat, timpul de execuţie este cel mai mare cu putinţă şi are valoarea:

$$T_{max}=T_{in}+Tv_1+Tc_1+Tv_2+Tc_2+\ldots+Tv_k+Tc_k+T_{out} = T_{in}+Tv_1+Tv_2+\ldots+Tv_k+T_{out}+ Tc_1+Tc_2+\ldots+Tc_k =$$

$$= T_{min}+ Tc_1+Tc_2+\ldots+Tc_k.$$

Din analiza anterioară rezultă că pot exista variaţii mari între timpul minim şi cel maxim de execuţie a buclei.

A doua modalitate de a măsura trecerea timpului este să executăm o buclă `for` cu corpul vid, cum ar fi `for(i=0; i<125000UL; i++){}`. Această metoda este mult mai precisă dar are dezavantajul că atât timp cât se numără nu se mai pot executa alte acţiuni deoarece toată forța de calcul a procesorului este folosită în ciclul `for`. Cu toate acestea metoda este uneori utilizată, în special pentru intervale mici de timp, de ordinul microsecundelor sau al zecilor de microsecunde.

Ambele metode de măsurare a timpului prezentate anterior prezintă dezavantaje: ori sunt foarte imprecise, ori consumă forţă de calcul. Rezolvarea acestei probleme se bazează pe modul în care oamenii execută acţiuni la anumite momente de timp: **ceasul cu alarmă** sau **temporizatorul cu alarmă**. De exemplu, pentru a încălzi mâncarea cinci minute la cuptorul cu microunde setăm temporizatorul la cinci minute iar când acesta ajunge la zero se va activa o alarmă sonoră. Putem executa orice altă acţiune cât timp funcționează temporizatorul dar când auzim alarma ne vom întrerupe activitatea curentă, vom executa acțiunea cerută de scurgerea timpului programat şi apoi ne vom relua activitatea întreruptă.

La microcontrolere rolul ceasului sau al temporizatorului este îndeplinit de numărătoare iar rolul alarmei este îndeplinit de sistemul de întreruperi. Întreruperile se vor trata într-o prelegere viitoare.

2. Structura generală a unui counter/timer

Măsurarea trecerii timpului necesită numărătoare. Un numărător binar este o maşină secvenţială Moore la care starea următoare se obţine din starea prezentă la care se adună sau se scade unu. Dacă se adună 1 spunem că numărătorul numără înainte iar dacă se scade 1 spunem că numărătorul numără înapoi. Schimbarea stării numărătorului are loc pe unul din fronturile semnalului de ceas (de regulă pe frontul ridicător). Astfel, numărătorul va avea:

- **intrare de ceas - CLK** (clock),
- ***n* ieşiri: Q[n-1:0]** şi
- intrare de **direcţie**. Această intrare se poate numi **UP** sau U/D. Pentru UP = ‚1' numărătorul numără în sens crescător iar pentru UP = ‚0', în sens descrescător.

În afară de intrarea de ceas CLK şi ieşirile Q, numărătorul mai poate fi încărcat paralel, la fel ca un registru. Pentru încărcare paralelă sunt necesare:

- ***n* intrări de date D[n-1..0]** şi o
- intrare de control a încărcării **LD – Load**. Ideea este să putem seta valoarea stării următoare a numărătorului indiferent de valoarea stării prezente. Setarea se face prin activarea intrării **LD**. Valoarea stării următoare se stabileşte prin intermediul a celor *n* intrări de date **D[n-1..0]**. De exemplu, dacă la un numărător pe 3 biţi se adaugă încărcarea paralelă se poate obţine secvenţa: …→ 000 → 001 → 101 → 110 →111 → 000 → ….Starea 101 s-a obţinut prin încărcare paralelă. Dacă încărcare paralelă nu ar fi fost activă pe starea 001, în loc de 101 starea următoare ar fi fost 010.

  Încărcarea paralelă permite generarea de intervale de timp programabile tip temporizator. De exemplu, dacă se doreşte ca aplicaţia să aştepte un interval de timp de 1 ms iar perioada ceasului CLK este de 1μs, se încarcă numărătorul cu valoarea 1000, numărătorul începe să numere înapoi şi după 1000 de impulsuri de ceas atinge valoarea zero. Între momentul încărcării şi momentul atingerii valorii zero trece 1 ms.

În afară de semnalele trecute în revistă anterior, un numărătorul mai poate fi prevăzut cu:

- Intrare de validare a numărării – **CE** (Clock Enable). Dacă CE este ‚0' numărătorul ignoră semnalul de ceas şi îşi conservă starea.
- Intrare de ştergere **CLR**-Clear. Dacă **CLR** este activ, starea numărătorului devine 0…000, adică toate ieşirile Q devin zero. Intrările CLR şi LD pot fi de tip sincron sau asincron, în funcţie de implementare.
- **TC** – Terminal Count. Este o ieşire activă pe durata stării finale. Dacă numărătorul numără înainte starea finală este starea în care toţi biţii numărătorului au valoarea ‚1' iar dacă numărătorul numără înapoi este starea în care toţi biţii sunt ‚0'. **Atenţie**: după ce un numărător atinge starea finală numărarea nu se opreşte ci continuă cu prima stare. De exemplu, un numărător binar pe doi biţi numără înainte astfel: …→ 00 → 01 → 10 → 11 → 00 → 01 → 10 → 11 → 00 →……. „00" este prima stare iar „11" este starea finală. Semnalul **TC** este activ o perioadă de ceas din 2^n perioade. În exemplul anterior TC este activ din 4 în 4 perioade de ceas. În momentul în care numărătorul trece din starea finală în prima stare spunem că numărătorul ciclează, adică începe un nou ciclu de numărare.

Numărătorul generic prezentat mai sus se conectează la un sistem cu microprocesor. Modul în care se face această conectare este detaliat în continuare.

1.1 Conectarea numărătorul cu procesorul

Numărătorul se mapează în spaţiul de adrese al procesorul la fel ca un registru paralel-paralel şi astfel numărătorul devine un port buffer. Portul buffer a fost prezentat în prelegerea 2, figura 1. Scrierea acestui port înseamnă încărcarea paralelă a numărătorului iar citirea înseamnă citirea stării numărătorului.

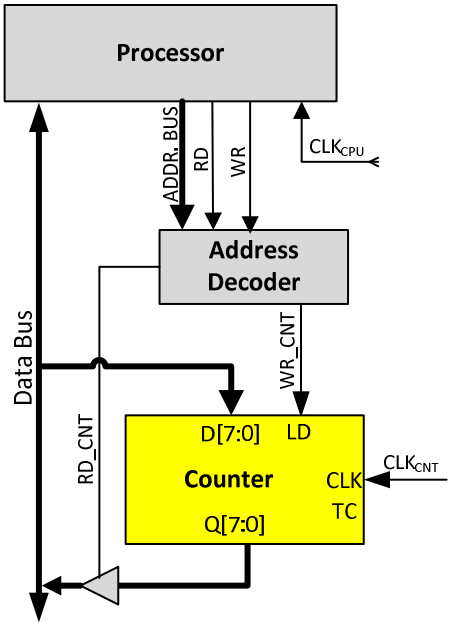


figura 1

În figura 1 este prezentată conectarea unui numărător pe 8 biți la un procesor cu magistrala de date tot de 8 biţi.

Decodificatorul de adrese ADDRESS DECODER generează semnalele WR_CNT şi RD_CNT. Semnalul WR_CNT se activează când procesorul execută o scriere la adresa alocată numărătorului iar semnalul RD_CNT se activează când procesorul execută o citire la adresa alocată numărătorului. Decodificarea adreselor şi decodificatorul de adrese sunt prezentate pe larg în prelegerile 1 şi 2.

Cele spuse anterior sunt adevărate dacă numărul de biţi ai numărătorului este egal cu lăţimea magistralei de date a procesorului. Deoarece ATmega16 are magistrala de date de 8 biţi, rezultă că numărătorul trebuie să aibă tot 8 biţi. Există însă multe aplicaţii pentru care 8 biţi nu sunt suficienţi. În acest caz numărătorul se va implementa pe 16. Un numărător pe 16 biţi va ocupa două adrese procesor.

1.2 Prescalerul

Următoarea secțiune a schemei de conectarea a numărătorului la procesor o constituie **secțiunea de generare şi selecţie a ceasului de numărare.**

Aşa cum s-a discutat anterior, măsurarea timpului scurs este o operaţie necesară în marea majoritate a aplicaţiilor de control. Intervalele de timp care trebuiesc măsurate variază de câteva microsecunde la minute sau ore. În cele ce urmează vom presupune că semnalul de ceasul microcontrolerului are frecvenţă de 10MHz (perioada de 100 ns), deoarece marea majoritate a microcontrolerelor au frecvenţa ceasului în jurul acestei valori. Un numărător pe 8 biţi evoluează în plaja 0 – 255 şi astfel intervalul maxim de timp care poate fi măsurat fără ca numărătorul să cicleze este 255*100ns=25,5 µs. Acuratețea măsurării este de un impuls de ceas, adică 100 ns.

Pentru a măsura intervale de timp mai mari există două soluţii: să mărim numărul de biți pe care se face numărarea la 16, 24 sau 32 de biţi sau să scădem frecvenţa ceasului numărătorului. Creşterea numărului de biţi pe care se face numărarea are dezavantajul creşterii complexităţii hardware-ului iar scăderea frecvenţei are dezavantajul scăderii preciziei. De exemplu, dacă am scădea frecvenţă ceasului de numărare de la 10 MHz la 5 MHz perioada ar creşte de la 100 ns la 200 ns. În acest caz precizia ar fi de 200 ns, adică la jumătate.

Soluţia aleasă de toţi producătorii de microcontrolere este **scăderea frecvenţei ceasului** numărătorului deoarece în majoritatea aplicaţii nu sunt necesare simultan o precizie mare şi un interval de timp mare. Ce sens ar avea ca timpul de spălare la o maşina de spălat sau timpul de încălzire la un cuptor cu microunde să fie măsurat cu o precizie de 100 ns? În acest caz o precizie de o zecime de secundă este mai mult decât suficientă.

Pentru scăderea frecventei, ceasul de numărare se obține se prin divizarea ceasului procesorului u o putere a lui 2, fiind uzuale frecvenţele $f_{CLK\_CPU}/8$, $f_{CLK\_CPU}/64$, $f_{CLK\_CPU}/256$, $f_{CLK\_CPU}/512$ şi $f_{CLK\_CPU}/1024$. Aceste frecvenţe nu sunt standard şi diferă de la producător la producător şi chiar de la microcontroler la microcontroler, chiar dacă sunt fabricate de acelaşi producător. Blocul care divizează ceasul procesor la 8, 64, …, 1024 se numeşte **prescaler**. Această denumire este folosită de toți producătorii de microcontrolere. În figura 2 este prezentat prescalerul precum şi celelalte blocuri din secțiunea de generare şi selecţie a ceasului de numărare.

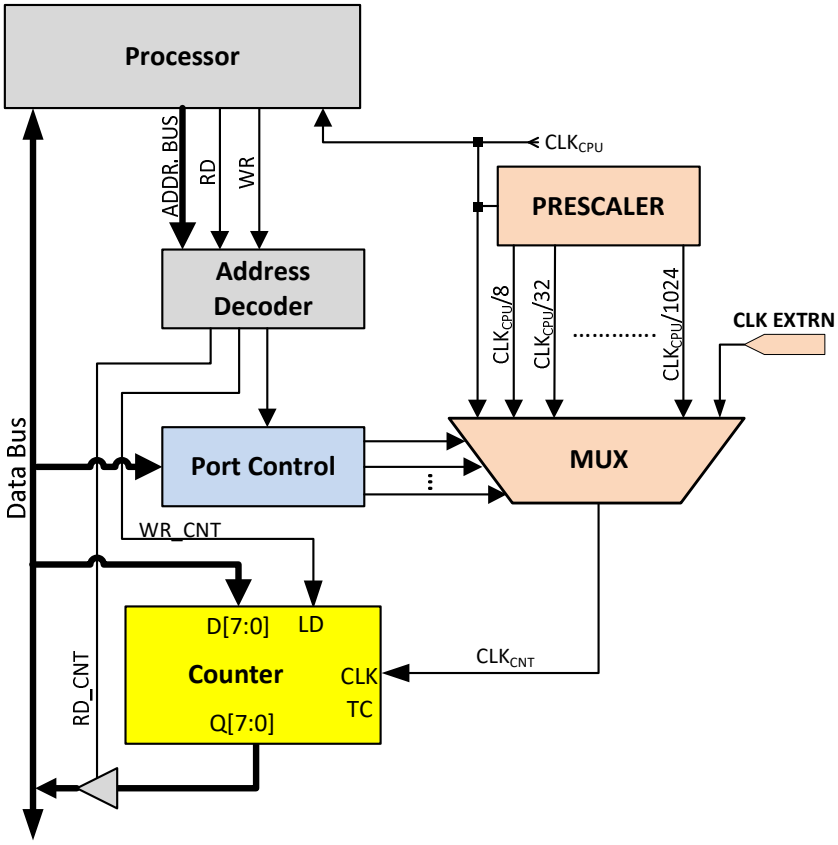


figura 2

Următorul element din **secțiunea de generare şi selecţie a ceasului de numărare** este un multiplexor care primeşte la intrare ceasul procesor plus ceasurile obţinute prin divizarea ceasului procesor şi generează la ieşire ceasul numărătorului, notat CLK_{CNT}. Acest multiplexor se numeşte MUX în figura 2. Intrările de selecție ale multiplexorului MUX sunt conectate la un port de ieşire, fiind astfel sub controlul software-ului. Portul de ieşire prin care se selectează ceasul de numărare constituie **portul de control al numărătorului** şi este denumit PORT CONTROL în figura 2. Pentru selecția ceasului de numărare nu sunt necesari toți biții portului de control; ceilalţi biţii vor controla funcţii ce vor fi discutate în scurt timp.

În afară de măsurarea scurgerii timpului mai există o operaţie care trebuie implementată în multe aplicații micro, şi anume **numărarea evenimentelor** externe. Un exemplu îl constituie numărarea persoanelor care au intrat într-un magazin. Pentru aceasta, la intrarea magazinului se montează un senzor care oferă un impuls la fiecare trecere a unei persoane. Acest impuls, numit în continuare **CLK EXTERN**, poate fi folosit de un numărător pentru contorizarea evenimentelor. Pentru această operaţie numărătorul trebuie să numere înainte. Semnalul CLK EXTERN necesită o intrare suplimentară în multiplexorul MUX din figura 2.

1.3 Indicatorul de ciclare (Timer overflow - TOV)

Următoarea operație implementată de toate numărătoarele din microcontrolere o constituie **extensia software a numărului de ranguri** pe care se face numărarea. Aşa cum s-a precizat anterior numărătorul numără pe 8 sau 16 biţi. În figura 1 numărătorul numără pe 8 biţi, ceea ce înseamnă că valoarea maximă pe care o poate atinge este 255. Pentru a putea număra mai mult de 255 de impulsuri de ceas software-ul trebuie să detecteze când s-a atins starea finală 255 şi apoi să incrementeze o variabilă. Această variabilă constituie extensia software a numărătorului. Astfel numărul de impulsuri de ceas va fi contorizat de variabila software concatenată cu numărătorul hardware.

Pentru ca lucrurile să fie mai clare se va considera **următoarea analogie**: dispunem doar de un **cronometru care indică numai secundele** şi se cere să măsurăm timpul în secunde, minute şi ore.

Cronometru este un ceas analogic care are numai secundar. Soluția evidentă în acest caz este să monitorizăm **continuu** cronometru şi la fiecare trecere a limbii prin 60 să notăm pe o hârtie că a mai trecut un minut. Pe hârtie putem să ținem evidenţa timpului scurs în minute şi ore iar secundele sunt indicate de cronometru.

Dezavantajul acestei soluţii este că necesită monitorizarea continuă a cronometrului, adică nu trebuie să ne dezlipim ochii de pe acesta. Evident, această soluţie este inacceptabilă. Soluţia corectă este să dotăm cronometrul cu **alarmă** şi să desfăşurăm alte activităţi în paralel cu numărarea. Când auzim alarma **întrerupem** activitatea curentă, actualizăm timpul pe hârtie şi apoi ne **reluăm** activitatea întreruptă. Alarma este un indicator acustic care devine activ când secundarul trece prin 60. Indicaţia acustică rămâne activă până când este apăsat un buton de anulare. Mai întâi ne vom ocupa de alarmă.

Numărătorul din microcontroler este cronometrul din exemplu anterior cu deosebirea că starea finală a cronometrului este 60 iar cea a numărătorului este 255 (dacă numărarea se face pe 8 biţi). În ceea ce priveşte indicatorul care precizează trecerea prin starea finală, numărătorul are o deja o ieşire numită TC – Terminal Count – care semnalizează ca numărătorul se află în starea finală. Descrierea ieşirii TC s-a făcut la începutul acestui capitol.

Problema cu TC este că acesta este activ doar pe starea finală, adică pe durata unui singur impuls de ceas, ceea ce este prea puțin. În analogia cu cronometrul cu alarmă este ca şi cum alarma ar fi activă doar o secundă. Cum timpul de execuţie al buclei principale `while(1)` poate fi şi de ordinul zecilor de milisecunde, generarea alarmei din TC necesită circuitele logice din figura următoare:

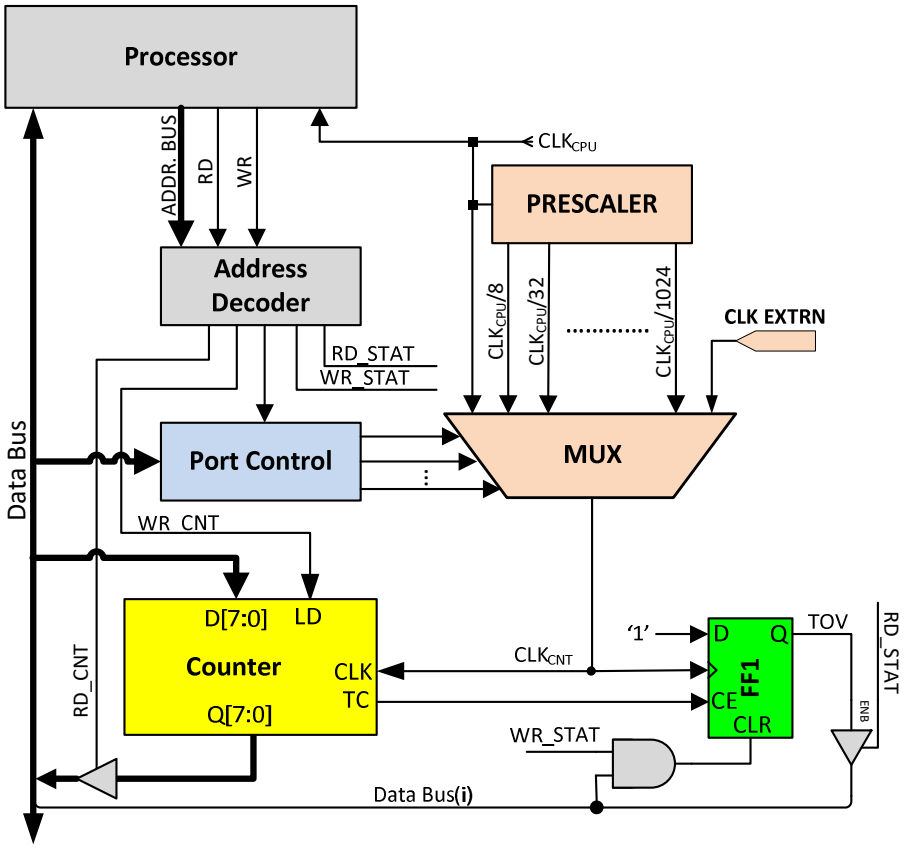


figura 3

Ieşirea TC a numărătorului se conectează la intrarea validare CE a unui bistabil care are intrarea D conectată la constanta ‚1' şi intrarea CLK la ceasul numărătorului. Când numărătorul ajunge în starea 255 semnalul TC devine activ astfel încât **următorul** front al ceasului va înscrie ‚1'-ul în bistabil. Ieşirea bistabilului va rămâne ‚1' indiferent de valoarea ulterioară a lui TC. Bistabilul care indică ciclarea este notat **TOV** (**T**imer **OV**erflow) şi referinţa lui este FF1 în figura 3. Acest bistabil este bitul *i* dintr-un port

de intrare/ieşire al microsistemului şi are alocată o adresă în spaţiul de adrese procesor. Acest port constituie portul de stare al numărătorului.

Decodificatorul de adrese ADDRESS DECODER generează semnalele RD_ STAT şi WR_STAT. Semnalul RD_STAT se activează când procesorul execută o citire de la adresa alocată portului de stare iar WR_STAT atunci când se execută scriere. De remarcat că scrierea bitului *i* din portul de stare nu se face în mod obișnuit, adică valoarea scrisă devine noua stare a bistabilului. Scrierea obișnuită se face prin intermediul intrării D a bistabilului, dar în cazul bistabilului FF1 intrarea D este conectată permanent la ‚1', după cum s-a discutat anterior. Deoarece intrarea D nu este liberă, semnalul WR_STAT va acţiona asupra intrării CLR – Clear. Dacă portul de stare ar fi alcătuit numai din bistabilul FF1, ar fi suficient să conectăm WR_STAT la intrarea CLR. Astfel scrierea portului de stare cu orice valoare ar duce la ştergerea lui FF1. Dar, aşa cum vom vedea în continuare, portul de stare mai conţine şi alt bistabil. În acest caz scrierea ar şterge ambele bistabile. Pentru a şterge un anume bistabil se impune o condiție suplimentară: bitul *i* din portul de stare se va şterge doar dacă este scris cu valoarea ‚1'. Adică FF1 se va şterge dacă simultan semnalul de scriere WR_STAT şi bitul *i* al magistralei de date sunt ‚1'. Această condiție este implementată prin intermediul porţii AND din figura 3. În concluzie ştergerea bitului *i* din portul de stare se face dacă valoarea scrisă în bitul *i* este ‚1'. În cazul în care se scrie ‚0', bitul *i* rămâne nemodificat.

În rezumat, soluţia problemei măsurării timpului cu cronometru are două componente: **alarmă** la cronometru şi **întrerupere** activităţii curente când se aude alarmă. Alarmă la numărător se numeşte TOV şi a fost implementată în figura 3 iar întreruperile vor fi tratate în prelegerea următoare.

Chiar dacă alarma TOV a fost implementată pentru a fi folosită de sistemul de întreruperi asta nu înseamnă că aceasta nu poate fi folosită şi în polling. TOV este un bit dintr-un port şi poate fi citit sau resetat în orice moment. Pentru ca pollingul să poată fi folosit **este obligatoriu** ca între două citiri ale numărătorului să nu treacă mai mult de un ciclu de numărare.

Bitul TOV nu este indispensabil pentru polling dar simplifică scrierea software-ului ce implementează metoda polling. **Ca o paranteză**, vom arăta cum se poate determina intervalul de timp între două evenimente cu un cronometru care are numai secundar şi alarmă la 60 de secunde. Vom considera următorul exemplu:

Eveniment:	e_0	e_1	e_2	e_3
Indicaţie cronometru:	0	20	50	30…
TOV:	0	0	0	1…
Minute pe hârtie:	0	0	0	1…
Interval între citiri:	0	20	30	40…

1. Cronometrul porneşte de la 0. Pornirea cronometrului este evenimentul e0.
2. Evenimentul e1 se produce la 20s. Intervalul de timp între e1 şi e0 este 20 – 0 = 20s.
3. Evenimentul e2 se produce la 50s. Intervalul de timp între e2 şi e1 este 50 – 20 = 30s.
4. Evenimentul e2 se produce la 30s. Dacă am calcula intervalul de timp între e2 şi e3 obţinem 30 – 50 = –20. Valoarea negativă indică ciclarea cronometrului. În cazul în care a avut loc ciclarea între cele două citiri intervalul de timp între citiri se calculează ca 30 + 60 – 50 = 40.

În cazul 4 se observă că ciclarea cronometrului poate fi detectată în două moduri: diferența de timp între două citiri succesive este negativă sau TOV este ‚1'. Software-ul poate detecta ciclare prin oricare din metode dar metoda cu TOV este un pic mai simplă. Până când va fi introdus sistemul de întreruperi vom folosi metoda TOV atât în curs cât și la laborator.

În final vom calcula durata ciclului de numărare. Vom relua analogia dintre un cronometru care are numai secundar şi numărător. Un ciclu al cronometrului înseamnă parcurgerea de către secundar a

întregului cadran. Durata unui ciclu este egală cu numărul de diviziuni ale cadranului înmulţit cu timpul necesar parcurgerii unei diviziuni. Cronometrul are cadranul împărţit în 60 de diviziuni iar parcurgerea unei diviziuni durează o secundă. Astfel un ciclu cronometru este egal cu 60*1s=60s=1min. Un numărător este un cronometru care are N diviziuni iar parcurgerea unei diviziuni este se data de perioada ceasului de numărare T_{CLK_CNT}. Durata unui ciclu numărător este T_{cycle}=N* T_{CLK_CNT}. Prescalerul generează ceasul de numărare T_{CLK_CNT} prin divizarea cu *p* a ceasului procesor T_{CLK_CPU}. Atunci putem scrie că

$$T_{CLK\_CNT} = p * T_{CLK\_CPU} \qquad \textbf{Rel. 1}$$

Dacă înlocuim pe T_{CLK_CNT} în durata ciclului de numărare T_{cycle}=N* T_{CLK_CNT} obţinem:

$$T_{cycle} = p * N * T_{CLK\_CPU} \qquad \textbf{Rel. 2}$$

Relația Rel. 2 este **relaţia fundamentală cu perioade** a numărătoarelor**.** Știind că T=1/*f* sau *f*=1/T putem ca în Rel. 2 să înlocuim perioadele cu frecvenţele corespunzătoare:

$$T_{cycle} = p * N * T_{CLK\_CPU}$$

$$\frac{1}{f_{cycle}} = p * N \frac{1}{f_{CLK\_CPU}}$$

În final obţinem:

$$f_{CLK\_CPU} = p * N * f_{cycle} \qquad \textbf{Rel. 3}$$

Relația Rel. 3 este **relaţia fundamentală cu frecvenţe** a numărătoarelor. În continuarea expunerii se vor folosi relaţiile 1-3.

Atenţie: Relaţiile 1-3 sunt adevărate numai dacă ceasul numărătorului CLK_CNT este un semnal periodic. CLK_CNT este periodic dacă se obţine prin divizarea cu *p* a ceasului procesor CLK_CPU. CLK_CNT poate să **nu** fie periodic dacă este generat extern (CLK EXTERN în figura 2 şi în figura 3).

3. Numărătorul (Timer/Counter) 0 la ATmega16

ATmega16 este prevăzut cu 2 numărătoare pe 8 biţi (0 şi 2) şi un numărător pe 16 biți. Oricare numărător (timer/counter în documentaţie) poate fi configurat să numere fie impulsuri interne (timer), fie impulsuri externe (counter). Cele trei numărătoare pot fi programate să funcţioneze în 4 moduri: normal, CTC, PWM rapid şi PWM corect. **Pe lângă informația care urmează este obligatoriu să parcurgeți din datele de catalog capitolele 14, 15, 16 şi 17.** Pentru început se va prezenta numărătorul 0.

3.1 Modul normal

Schema numărătorului 0 în modul normal este prezentat în figura următoare:

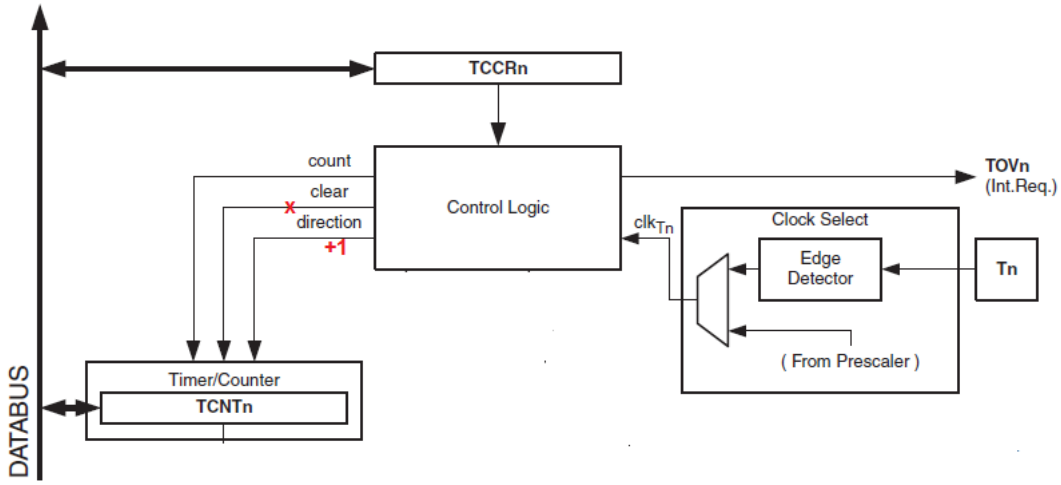


figura 4

În figură s-au reprezentat numai blocurile active în modul normal. Schema este foarte asemănătoare cu schema de principiu din figura 3. Blocurile care apar în ambele figuri sunt numărătorul, prescalerul, portul de control TCCR0 (Timer/Counter Control Register) şi indicatorul TOV0. De remarcat că indicatorul TOV a fost absorbit în blocul de control „Control Logic”. Ceasul numărătorului este generat de blocul „Clock Select”. Prescalerul este comun numărătoarelor 0 şi 1 şi este prezentat în figura următoare.

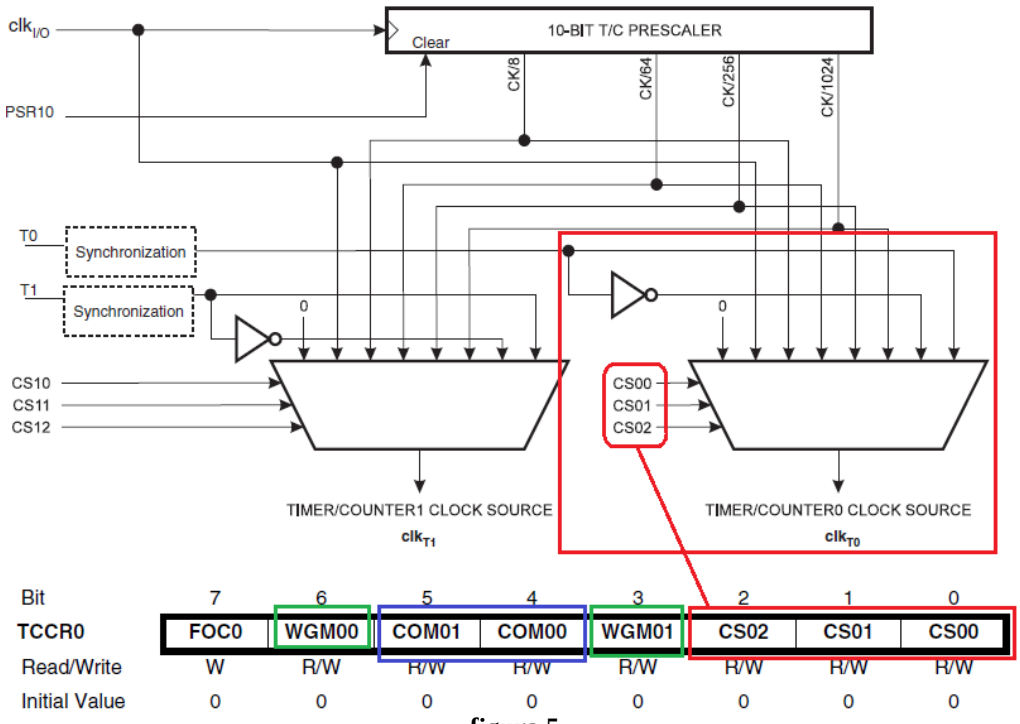


figura 5

Ceasul numărătorului 0 este generat la ieşirea multiplexorului din dreptunghiul roşu. Prin intermediul acestui multiplexor se poate selecta fie ceas extern, fie un ceas intern derivat din ceasul procesor. În esenţa acest multiplexor este multiplexorul MUX din figura 3, dar cu mai multe opţiuni. Prin intermediul a trei biţi din TCCR0 (registrul de control al numărătorului 0) – CS00, CS01 şi CS02 - ceasul numărătorului 0 poate fi ceasul procesorului, ceasul procesorului divizat cu 8, 64, 256 sau 1024, frontul ridicător sau frontul coborâtor al semnalului de numărare extern T0 sau constanta ‚0’. Selectarea constantei zero înseamnă numărător oprit.

Modul în care funcționează numărătorul 0 se selectează prin intermediul biților WGM01 şi WGM00 din TCCR0. Aceşti biţi sunt marcați cu verde în figura 5. Pentru modul normal aceştia trebuie să fie 0. În toate modurile de operare se definesc constantele BOTTOM, TOP şi MAX TOP (pe scurt MAX). BOTTOM ca cea mai mică valoarea pe care o poate avea numărătorul şi are valoarea 0x00. MAX

ca cea mai mare valoarea pe care o poate avea numărătorul şi are valoarea 0xFF. Valoarea TOP depinde de modul în care funcţionează numărătorul.

Biţii COM01 şi COM00 vor fi discutaţi în capitolul următor.

Modul normal este cel mai simple mod de operare. În acest mod numărătorul numără liber, adică **este un numărător modulo 256**. Direcția de numărare este numai înainte, fără ca să se execute vreodată ştergerea numărătorului. Starea curentă a numărătorului se obţine din starea anterioară la care se adună 1, cu o excepție: după starea MAX (0xFF) urmează starea BOTTOM (0x00). În modul normal indicatorul TOV0 (Timer/Counter Overflow Flag) va fi setat în acelaşi ciclu de ceas în care numărătorul devine 0. În acest caz TOV0 se comportă ca un al nouălea bit, cu excepția faptului că este doar setat în hardware, fără a fi şi şters. Ştergerea se poate face de software. Indicatorul TOV0 este disponibil prin intermediul bitului 0 din registrul de stare TIFR. Acest bit este marcat cu roşu în figura următoare:

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

figura 6

Modul normal este folosit frecvent pentru contorizarea evenimentelor externe. În continuare se va prezenta structura acestei aplicaţii.

3.2 Aplicaţie de contorizare evenimente externe cu Timerul 0 în modul normal

Pentru contorizarea evenimentelor externe este nevoie de un semnal care să îşi schimbe valoarea la apariția unui evenimentului. Numărătorul va contoriza numărul de fronturi ale respectivului semnal. Acest semnal trebuie aplicat pe pinul T0 al microcontrolerului ATMega16. Pentru împachetarea DIP40 pinul T0 este pinul 1. Acest pin trebuie programat ca intrare, adică DDRB(0) trebuie să fie ‚0'.

Este posibil ca să se contorizeze fie fronturile ridicătoare, fie fronturile coborâtoare ale semnalului de numărare. Pentru ca numărarea să se facă pe front ridicător biţii CS02:CS00 trebuie să aibă valoarea „111" iar pentru frontul coborâtor valoarea „110".

După cum s-a precizat anterior numărul de biţi pe care se face numărarea trebuie extins în software deoarece numărătorul numără doar pe 8 biţi. Extensia software se face cu ajutorul bitului de stare TOV0: când numărătorul ciclează, adică trece din starea 255 în starea 0, bitul TOV0 este setat automat de hardware. Deci de fiecare data când TOV0 este ‚1' înseamnă că s-au mai numărat 256 de evenimente. Astfel numărul de evenimente apărute este dat de:

numărul de apariții ale lui TOV0 *256 + valoarea curentă a numărătorului.

În program trebuie să monitorizăm pe TOV0 şi când acesta devine ‚1' vom incrementa o variabilă numită de exemplu `cnt_hi`. Apoi vom şterge imediat pe TOV0 pentru a captura următoarea tranziţie. Vom presupune că evenimentele ce se contorizează apar suficient de rar astfel încât TOV0 să fie setat cel mult o dată pe durata de execuție a unei iteraţii a buclei `while(1)`. Într-o primă aproximație extensia software se implementează prin intermediul următorului program:

```
//Application – Event Counter
include <avr/io.h>
int main(){

   unsigned char cnt_hi=0;
   unsigned int  cnt;

   //        ┌──┬Normal mode
   TCCR0=0b00000111;
   //        ||  └┴┴External clock source on T0 pin, ↑
   //        └┴────Normal port operation, OC0 disconnected.
   TCNT0=0;

   while(1){

      if(TIFR & 1<<TOV0){  // TOV0 is an alias for integer 0 in avr/io.h
         TIFR |= 1<<TOV0;  // Clear TOV0
         cnt_hi++;
      }

      cnt = cnt_hi*256 + TCNT0;

      //use cnt

   }//end while
}//end main
```

Programul de mai sus pare corect dar de fapt există o situaţie în care greşeşte. Această situaţie este următoarea:

```
while(1){
   //cnt_hi=0x02, TCNT0=0xFF, TOV0=0
   if(TIFR & 1<<TOV0){
      TIFR...
   }
   //cnt_hi=0x02, TCNT0=0x00, TOV0=1
   cnt = cnt_hi*256 + TCNT0;
```

Să presupunem că înainte de executarea instrucţiunii *if* variabila `cnt_hi=0x02, TCNT0=0xFF` şi `TOV0=0`. Deoarece TOV0 este ‚0', corpul *if*-ului nu se execută. Dar imediat după *if* apare un front ridicător al lui T0 şi numărătorul trece din 0xFF în 0x00 iar TOV0 devine ‚1'. Deoarece suntem exact după *if*, `cnt_hi` rămâne nemodificat; acesta va fi incrementa la următoarea iteraţie a buclei `while(1)`.

În concluzie, înainte de a calcula pe `cnt` avem `cnt_hi=0x02, TCNT0=0x00` şi `TOV0=1`. Cu aceste valori obţinem `cnt=2*256+0`. Această valoare este evident greşită, valoarea corectă fiind `cnt=3*256+0`. Această situaţie a apărut deoarece incrementarea numărătorului şi executarea programului sunt **două procese independente ce se execută în paralel**. Schimbarea valorii numărătorului se poate face în timpul execuţiei oricărei instrucţiuni din program.

Soluţia evidentă în acest caz este să ţinem seama de valoarea lui TOV0 atunci când calculăm valoarea lui `cnt`:

```
      if(TIFR&1)
         cnt = (cnt_hi+1)*256 + TCNT0;
      else
       //cnt_hi=0x02, TCNT0=0x00, TOV0=1
         cnt = cnt_hi    *256 + TCNT0;
```

Din păcate nici această modificare nu rezolvă problema deoarece numărătorul se poate incrementa exact înainte de calcularea lui `cnt` pe ramura *else*. Pentru a corecta aceasta deficienţă vom face o copie a numărătorului înainte de instrucţiunea *if*. Cu aceste două modificări programul de contorizare evenimente este următorul:

```
//Application – Counter
int main(){
   unsigned char cnt_hi=0, cnt_lo_copy;
   unsigned int  cnt;

   TCCR0=0b00000111;    TCNT0=0;
   while(1){
      //t1: //cnt_hi=0x02, TCNT0=0xFF, TOV0=0
      if(TIFR & 1){  //TIFR bit 1 is OCF0
         TIFR |= 1;
         cnt_hi++;
      }
      //t2
      cnt_lo_copy = TCNT0;
      //t3
      if(TIFR&1)
         cnt = (cnt_hi+1)*256 + TCNT0;
      else
         //t4
         cnt = cnt_hi * 256 + cnt_lo_copy;

     //use cnt

   }//end while
}//end main
```

Incrementarea numărătorului, adică situaţia `cnt_hi=0x02, TCNT0=0x00` şi `TOV0=1` poate apare în oricare din momentele de timp t_2, t_3 sau t_4. Dacă incrementarea apare la t_2 sau la t_3, atunci când se execută *if*-ul TOV0 va fi ‚1' şi valoarea calculată a lui `cnt` va fi cea de pe ramura *then*. Această valoare este corectă deoarece tine seama de faptul că TOV este ‚1'.

Dacă atunci când se execută *if*-ul TOV0 este ‚0' atunci calculul lui `cnt` se face pe ramura *else*. Pentru a anula efectul unei posibile incrementări a numărătorului la t_4 `cnt` se calculează cu copia numărătorului, copie care a fost făcută când valoarea lui TOV0 era 0.

În concluzie programul de mai sus calculează întotdeauna corect `cnt`.

3.3 Modul CTC - Clear Timer on Compare Match

Anterior s-a făcut o analogie între numărător şi un cronometru care indică doar secundele. Deosebirea consta în valoarea stării finale: starea finală a cronometrului este 60 iar cea a numărătorului este 255 (dacă numărarea se face pe 8 biţi). Se spune că numărătorul numără modulo 256 iar cronometru modulo 60.

În multe aplicaţii este necesară executarea periodică a anumitor acțiuni. Astfel de aplicaţii provin din domeniul transmisiei datelor. Problema în astfel de aplicaţii este că periodicitatea acțiunilor nu coincide cu perioada ciclării numărătorului. Mai întâi vom stabili care este perioada ciclării numărătorului, adică perioada semnalului TC. Secvența de numărare este următoarea:

```
Numărător:  0, 1, 2,……,254, 255, 0, 1, 2,……,254, 255, 0, 1, 2,……,254, 255,..
TC:         0, 0, 0,……,  0,   1, 0, 0, 0,……,  0,   1, 0, 0, 0,……,  0,   1,..

TOV:        0, 0, 0,……,  0,   0, 1, 1, 1,……,  0,   0, 1, 1, 1,……,  0,   0,..
```

Analizând secvenţă de mai sus se observă că TC apare pe starea finală 255. De la o starea 255 până la următoarea stare 255 numărătorul trebuie să numere 256 de impulsuri de ceas CLK_{CNT}. N-ul timer/counterului 0 în modul normal este 256. Dacă în Rel. 2 înlocuim pe *N* cu 256, obținem perioada lui TC şi implicit a lui TOV:

$$T_{cycle} = p * N * T_{CLK\_CPU} = 256*p* T_{CLK\_CPU}$$

Aşa cum s-a precizat anterior, această perioadă nu este adecvată într-o mulţime de aplicații. „Nu este adecvată" nu înseamnă că numărarea modulo 256 nu poate fi folosită, numai că necesită

monitorizarea continuă a numărătorului. De exemplu, dacă acțiunile cerute de aplicație trebuie executate din 50 în 50 impulsuri de numărare CLK_{CNT} atunci aceste acţiuni trebuie executate pe stările 49, 99, 149, 249, 43, etc. Cum TOV se activează doar pe starea 0, acesta nu poate fi folosit ca indicator şi nu ne mai rămâne decât să monitorizăm continuu numărătorul. Monitorizarea continuă înseamnă că nu mai putem executa şi alte acțiuni, aşa că această soluţie este rareori acceptată.

În acest caz soluţia este ca numărătorul să nu mai numere modulo 256 ci modulo *N*, valoarea *N* fiind programabilă. Pentru exemplu anterior, secvenţa de numărarea modulo 50 este:

```
Numărător:  0, 1, 2,……,48, 49, 0, 1, 2,……,48, 49, 0, 1, 2,……,48, 49, ..
EQ:         0, 0, 0,……, 0,  1, 0, 0, 0,……, 0,  1, 0, 0, 0,……, 0,  1, ..
```

Pentru a putea număra modulo *N* programabil adăugăm un comparator, un registru plus alte elemente ce vor fi discute imediat. În datele de catalog ale microcontrolerului modul de numărare modulo *N* se numeşte CTC – **C**lear **T**imer on **C**ompare Match. Timerul 0 va funcţiona în modul CTC dacă în portul de control biţii de mod WGM0(1:0) = „10”.

Componentele timerului 0 active în modul CTC sunt prezentate în figura următoare:

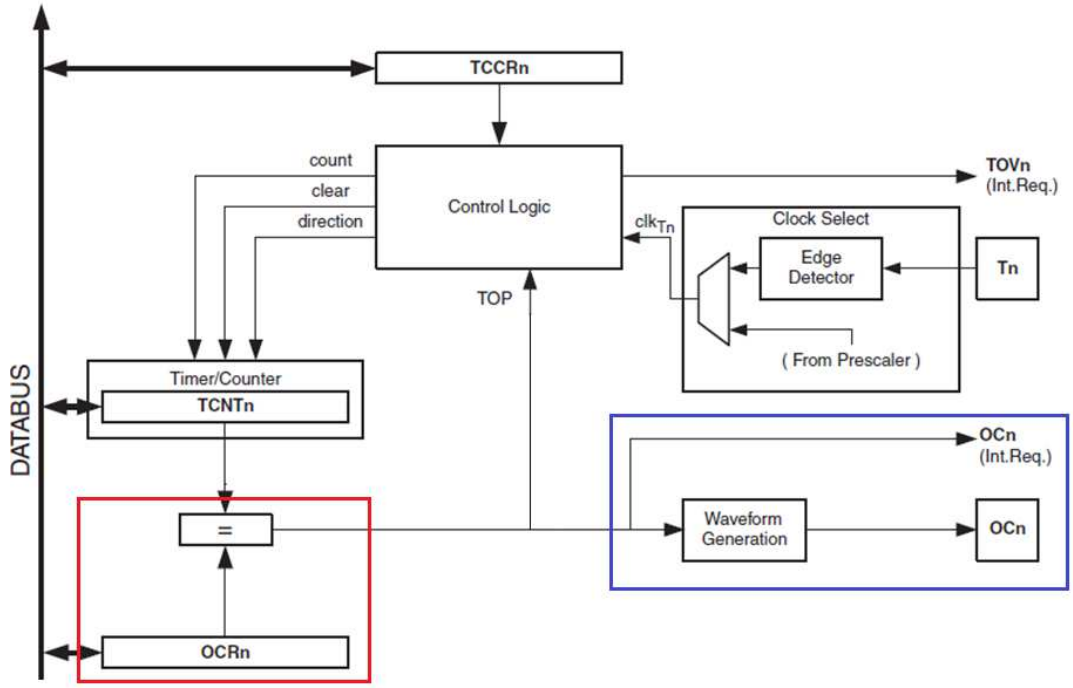


figura 7

1. Registrul în care se memorează valoarea maximă din secvenţa de numărare se numeşte **OCR0**. Acest registru este conectat la procesor ca port buffer. Valoarea scrisă în acest registru este comparată cu starea numărătorului de către comparatorul de egalitate „=” în figura 7. OCR0 şi comparatorul de egalitate sunt încadrate de dreptunghiul roşu din figura 7.
2. Ultimul element al modului de numărare modulo *N* variabil îl constituie blocul de semnalizare a egalităţii. Acest bloc are aceeaşi structura ca blocul care semnalizează TOV şi se numeşte **OCF0** (Output Compare Flag). Acest indicator va fi setat în ciclu de ceas ce urmează după detectarea egalităţii dintre starea numărătorului şi registrul OCR0. Indicatorul OCF0 poate fi doar setat de hardware. Ştergerea se poate face de software prin înscrierea cu ‚1’. Indicatorul OCF0 este accesat prin intermediul bitului 1 din registrului TIFR. Acest bit este marcat cu verde în figura 6.

În modul CTC numărătorul numără înainte până când starea sa devine egală cu conținutul registrului OCR0. După ce are loc egalitatea, pe următorul front al ceasului numărătorul este şters şi ajunge în starea BOTTOM (0x00). Conţinutul registrului OCR0 defineşte valoarea maximă până la care se numără şi se numeşte valoarea TOP în datele de catalog. Astfel numărătorul 0 va număra în mod CTC astfel: 0, 1, 2,

TOP-1, TOP, 0, 1, …. În acest mod numărătorul 0 devine un numărător modulo TOP+1. Cum valoarea TOP este programabilă, în modulul CTC numărătorul 0 este un numărător modulo programabil. O aplicaţie a timerului 0 în mod CTC va fi prezentată în capitolul următor.

În plus în figura 7 apare un bloc nou numit generator de formă de undă. Generatorul de formă de undă este încadrat în dreptunghiului albastru din figura 7. **Blocul de generare a formei de undă controlează ieşirea Output Compare** (OC0/PB3) disponibila pe pinul 4 la ATmega16. Prin intermediul blocului de generare a formei de undă şi a pinului OC0 activitatea timerului zero se poate vedea în exterior direct, fără intervenția software-ului, ceea ce micşorează timpul de răspuns al sistemului.

Blocul de generare a formei de undă poate fi configurat în funcţie de biţii COM01 şi COM00 din TCCR0. Aceşti biţi sunt marcaţi cu albastru în figura 5. Semnificaţia biţilor COM01 şi COM00 din TCCR0 este diferită în funcție de modul în care funcționează timerul. Dacă **modul de lucru este normal sau CTC**, atunci semnificaţia acestor biţi este:

1. COM0(1:0)=00. OC0 este **deconectat.** OC0 este controlat de PORTB.
2. COM0(1:0)=01. OC0 îşi schimbă valoarea (**Toggle**) când comparatorul detectează egalitate între starea numărătorului şi registrul OCR0.
3. COM0(1:0)=10. OC0 este şters (**Clear**) când comparatorul detectează egalitate între starea numărătorului şi registrul OCR0..
4. COM0(1:0)=11. OC0 este setat (**Set**) când comparatorul detectează egalitate între starea numărătorului şi registrul OCR0.

Pentru ca pinul OC0/PB3 să fie sub controlul blocului de generare a formei de undă trebuie ca biţii COM0(1:0)≠00 şi direcţia lui PB3 să fie out (DDR3=,1').

3.4 Ceas digital cu timerul 0

În mule aplicaţii trebuie să ţinem evidenţa timpului. Un ceas ține evidența timpului scurs în ore, minute şi secunde, un cronometru ţine şi evidenţă zecimilor de secundă şi există situaţii când evidenta trebuie ţinută în unități de 0,2 sau 0,5 secunde. Astfel **rezoluţia** cu care se măsoară timpul poate fi o secundă, o zecime de secundă, 0,2 secunde etc.

Obţinerea acestei rezoluții se face cu un semnal periodic cu perioada T_r (r de la rezoluţie). În mod ideal, T_r se generează cu un timer/counter care ciclează exact la T_r, adică $T_r = T_{cycle}$. Din păcate perioada T_{cycle} cu care ciclează un timer pe 8 biţi este de cele mai multe ori mai mică decât T_r. Conform relaţiei Rel. 2, $T_{cycle} = p * N * T_{CLK\_CPU}$. Cel mai mare p pentru un timer din ATmega16 este 1024 iar cel mai mare N pentru un timer pe 8 biţi este 256. Referitor la T_{CLK_CPU}, dacă se doreşte forța maximă de calcul, şi de regulă se doreşte, trebuie aleasă frecvenţa maximă a ceasului procesor, şi anume 16MHz. Pentru f_{CLK_CPU}=16MHz obţinem $T_{cycle} = p * N * T_{CLK\_CPU}$ = 1024 * 256 *1/16MHz = 16,39 ms. Evident, această valoare este mai mică decât rezoluţiile uzuale de 0,1, 0,5 sau 1s discutate anterior.

Deoarece $\mathbf{T_r \geq T_{cycle}}$, vom forma T_r din mai multe cicluri de numărare T_{cycle}. Astfel relaţia fundamentală pentru măsurarea timpului cu numărărtoare este:

$$T_r = kT_{cycle},\ k=1, 2, 3\ldots \qquad \textbf{Rel. 4}$$

Practic vom incrementa variabila software k la fiecare ciclare a numărătorului şi când aceasta ajunge la valoarea k ştim că a trecut timpul T_r.

Aplicaţia care arată cum se utilizează timerul 0 pentru măsurarea timpului este un ceas digital la care separatorul dintre ore şi minute clipeşte, fiind jumătate de secundă stins, jumătate aprins. Rezultă că

T_r este 0,5 secunde deoarece la fiecare jumătate de secundă trebuie să se execute o acţiune. Un astfel de ceas digital se poate folosi la cuptorul cu microunde la mașina de spălat, sau ca ceas de masă. În continuare vom arăta cum se obţine **T_r= 0,5** sec urmând ca să implementăm orele, minutele şi secundele la laborator.

Frecvență ceasului procesor pentru aplicaţia ceas digital este **9,216 MHz**. În mod uzual frecvenţa ceasului procesor este frecvenţă maxim posibilă pe care o suportă microcontrolerul, adică 16 MHz în cazul ATmega. Această frecvenţă se obţine însa numai cu un cristal de cuarţ extern, ceea ce înseamnă un preț de cost mai mare. Frecvența maximă care se poate obţine cu oscilatorul intern din ATmega (fără componente externe) este 8MHZ. Pe bună dreptate se pune întrebarea de ce s-a ales frecvenţa de 9,216 MHz. Această frecvenţă este necesară interfeţei seriale: pentru vitezele standard de emisie şi de recepţie ceasul procesor trebuie să fie un multiplu de 16*115200. Frecvenţă 9,216Mhz = **480***16*115200 Hz este suficient de mare pentru o forţă de calcul decentă şi este şi multiplu de 16*115200.

În cele ce urmează numărătorul 0 va funcţiona în modul CTC. În modul CTC numărătorul 0 este numărător modulo N, adică numără 0, 1, 2, N-1, 0, 1, … , N-1, 0,… Parcurgerea stărilor de la 0 la N-1 constituie un ciclu de numărare. Într-un ciclu numărătorul numără *N* de impulsuri ale ceasului de numărare CLK_{CNT}. Valoarea N-1 este valoarea care trebuie înscrisă în OCR.

Pentru a obţine Tr=0,5 s vom folosi relaţiile Rel. 3 şi Rel. 4. Mai întâi vom exprima Rel. 4cu frecvenţe:

$$T_r = k\,T_{cycle},\ T = \frac{1}{f} \quad \Rightarrow \quad \frac{1}{f_r} = \frac{k}{f_{cycle}} \quad \Rightarrow \quad f_{cycle} = k\,f_r$$

Dacă înlocuim f_{cycle} din relaţia de mai sus în relația Rel. 3, rezultă:

$$f_{CLK\_CPU} = pNf_{cycle},\ f_{cycle} = k\,f_r \quad \Rightarrow \quad f_{CLK\_CPU} = pNk\,f_r \quad \Rightarrow \quad \frac{f_{CLK\_CPU}}{f_r} = kpN$$

În concluzie putem scrie:

$$\frac{f_{CLK\_CPU}}{f_r} = kpN \qquad \textbf{Rel. 5}$$

În relaţia Rel. 5 folosim f_{CLK_CPU}=9,216MHz şi f_r=1/0,5s=2Hz:

$$\frac{f_{CLK\_CPU}}{f_r} = \frac{9{,}216MHz}{2Hz} = 9216000/2 = (2^{13}\ 3^2\ 5^3)/2 = 2^{12}\ 3^2\ 5^3$$

În relaţia Rel. 5 vom căuta cel mai mic *k*, ceea ce este echivalent cu căutarea celui mai mare produs *p*N. Suplimentar, *p* trebuie să fie egal cu 1, 8, 64, 256 sau 1024 iar *N* să fie mai mic ca 256:

$$k\,p\,N = 2^{12}\ 3^2\ 5^3 = 2 * 3^2 * 2^{10} * 2 * 5^3 \rightarrow k = 2 * 3^2 = 18,\ p = 2^{10},\ N = 2 * 5^3 = 250 < 256$$

Motivul pentru care se doreşte cel mai mic *k* va fi precizat în scurt timp.

Programul care implementează contorizarea ceasul cu clipirea separatorului cu timerul 0 este:

```
int main(){
   unsigned char cycles=0, one_sec=0, sec=0;

   //        ┌──┬CTC mode
   TCCR0=0b00001101;
   //        ||  └┴┴P=1024
   //        └┴──────Normal port operation, OC0 disconnected.
   OCR0=250-1; // N=250
```

```
   while(1){
      //...
      //t1
      if(TIFR & 1<<OCF0){ // OCF0 este bitul 1 din TIFR. #define OCF0 1 in avr/io.h
         TIFR |= 1<<OCF0; // clear OCF0
         cycles++;

         if(cycles == 18){ k=18
            // a trecut jumătate de secundă
            cycles=0;
            // aprinde sau stinge separatorul
            one_sec++;

            if(one_sec==2){
               one_sec=0;
               // a trecut o secundă
               // actualizează secunde, minute, ore
               // foloseşte timpul (de exemplu afişează-l)
            }
         }
      }
      //t2
      //...
   }//end while
}//end main
```

În acest moment vom explica de ce **se doreşte o valoare cât mai mică pentru *k***. Din relaţia Rel. 4 $T_r = k * T_{cycle}$ rezultă $T_{cycle} = T_r/k$. Cu cât *k* este mai mare, cu atât T_{cycle} este mai mic. Indicatorul OCF0 este setat de hardware la fiecare ciclu de numărare. Dacă activarea lui OCF0 se detectează prin **polling**, în bucla principală *while*(1) trebuie să se sesizeze dacă OCF0 a devenit 1. Rezultă că timpul maxim de execuţie a buclei *while*(1) trebuie să fie mai mic decât T_{cycle}, adică $T_{max\_while(1)} < T_{cycle} = T_r/k$. Un *k* mare înseamnă un $T_{max\_while(1)}$ mic iar un $T_{max\_while(1)}$ mic înseamnă imposibilitatea de a face toate prelucrările cerute de aplicaţie. Pentru a avea un T_{cycle} **cât mai mare** avem nevoie de un *k* mic.

Pentru a avea timpi de execuţie a buclei principale $T_{max\_while(1)} > T_{cycle}$ vom renunţăm la polling şi vom folosim întreruperile, aşa cum se va arăta în prelegerea următoare. Chiar dacă se folosesc întreruperile, un *k* mare înseamnă că vom procesorul va fi întrerupt de multe ori şi se va pierde forţă de calcul. Şi în cazul întreruperilor este de dorit un *k* mic.

În exemplul anterior T_r=0,5 s iar *k*= 18. Rezultă $T_{cycle} = T_r/k$ = 0,5 s/18=0,027 s = 27 ms. Această valoare este perfect acceptabilă deoarece timpul de execuţie a buclei principale în majoritatea aplicaţiilor se doreşte a fi în jurul valorii de 10 ms. În multe cazuri însă valoarea lui T_{cycle} este prea mică şi nu poate fi acceptată. Un astfel de caz este T_r=0,5s, la fel ca în exemplul anterior, dar f_{CLK_CPU} este 10 MHz.

Prin aplicarea relaţiei Rel. 5 pentru T_r=0,5s şi f_{CLK_CPU} =10 MHz obţinem:

$$\frac{f_{CLK\_CPU}}{f_r} = \frac{10MHz}{2Hz} = 10^7/2 = (2^7 * 5^7)/2 = 2^6 * 5^7$$

$$p * N * k = 2^7 * 5^7 = 2^6 * 5^3 * 5^4 \rightarrow p = 2^6,\quad N = 5^3 = 125 < 256,\quad k = 625$$

S-a obţinut cel mai mare produs *p**N dar *k*=625 ceea ce înseamnă $T_{cycle} = T_r/k$ = 0,5 s/625 =0,8ms, ceea ce este foarte puţin. Soluţia ar fi să folosim timerul 1 pentru care N maxim este 65536. Implementarea cu timerul 1se face ulterior în capitolul dedicat timerului 1.

În final trebuie analizată performanţă implementării de tip polling. În acest caz prin performanţa se înţelege eroarea metodei. **Eroarea** se calculează pe baza diferenţei dintre valoarea măsurată şi valoarea reală a cantităţii ce se măsoară. Evident, eroarea nu poate fi mai mică decât rezoluţia: un ceas digital care afişează orele şi minutele are o rezoluţie de un minut. Valoarea minutelor este aceeaşi şi pentru 0 secunde, şi pentru 59 de secunde. În concluzie eroarea maximă este cel puţin 59 de secunde.

În cazul numărătoarelor hardware valoarea măsurată se exprimă în T_{cycle}, aşa că rezoluţia este cel puţin un T_{cycle}, deci eroarea nu poate fi mai mică decât T_{cycle}. În cazul polling eroarea creşte datorită **întârzierii** dintre momentul în care numărătorul ciclează şi momentul în care timpul măsurat este folosit de software. Momentul în care timpul măsurat este folosit de software este marcat cu **`//foloseşte timpul (de exemplu afişează-l)`** în programul de la pagina 16.

În programul de la pagina 16 cea mai **mică întârziere** Δ_{min} se obţine dacă numărătorul ciclează exact înainte de momentul în care este testat OCF0. Acest moment este marcat cu //**t1** în programul de la pagina 16. Δ_{min} este timpul scurs între //**t1** şi **`//foloseşte...`**.

Cea mai **mare întârziere** Δ_{max} se obţine dacă numărătorul ciclează exact după de momentul în care este testat OCF0. Acest moment este marcat cu //**t2** în programul de la pagina 16. Δ_{max} este timpul de execuţie a buclei while(1) plus întârzierea Δ_{min} discutată mai sus. În concluzie eroarea măsurării timpului în polling este:

$$err_{poll} = T_{cycle} + \Delta_{max} = T_{cycle} + T_{max\_while(1)} + \Delta_{min} \qquad \textbf{Rel. 6}$$

Nu uităm că metoda polling poate fi folosita numai dacă $T_{max\_while(1)} < T_{cycle}$, după cum s-a arătat anterior. Dacă însă folosim întreruperile în loc de polling, această restricţie dispare iar eroarea va scădea, aşa cum se va arăta în prelegerea următoare.

3.5 Modul fast PWM – fast Pulse Width Modulation

Mai întâi se va explica ce este PWM. Pentru aceasta vom considera programul *blink* din laboratorul 2 care făcea să clipească un LED. Acest program a fost reluat la începutul acestei prelegeri, la pagina 1.

LED-ul conectat pe bitul 0 al portului A va executa la infinit un ciclu aprins-stins. Vom nota cu t_A timpul cât LED-ul este aprins, cu t_S timpul cât LED-ul este stins şi cu t_C=t_A+t_S durata unui ciclu aprins-stins. Pe baza timpilor t_A, t_S şi t_C vom defini mărimea numită factor de umplere (**duty cycle** sau **duty factor** în engleză). Această mărime exprimă în procente durata cât LED-ul luminează raportată la durata ciclului aprins-stins. Factorul de umplere se calculează cu formula:

$$D = \frac{t_A}{t_C} 100$$

În exemplu iniţial din laboratorul 2 timpul t_A este egal cu timpul t_S, ceea ce face ca factorul de umplere D să fie 50% ($D = \frac{t_A}{t_C}100 = \frac{t_A}{t_A + t_S}100 = \frac{t_A}{t_A + t_A}100 = 50\%$).

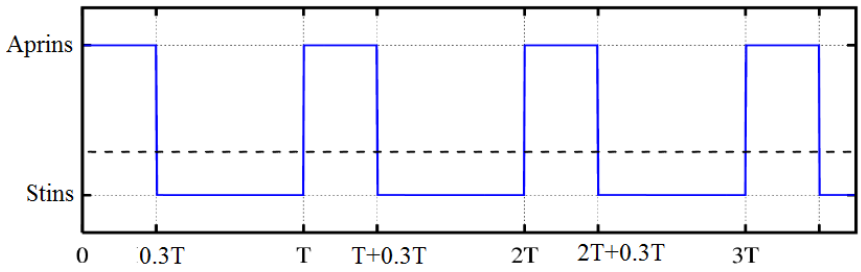
Un fenomen interesant apare dacă reducem durata ciclului aprins-stins de la o secundă la 10 milisecunde. În acest caz ochiul nu va mai percepe că LED-ul este aprins şi apoi stins ci va avea senzaţia că LED-ul luminează la jumătate din intensitate. Mai exact, dacă un LED tot timpul aprins luminează cu intensitatea L_{ON} iar un LED tot timpul stins cu luminozitatea zero, LED –ul care luminează într-un ciclu infinit aprins-stins cu factorul de umplere 50% va crea senzaţia că luminează cu intensitatea L=L_{ON}/2. Pentru ca să apară această senzaţie durata t_C a ciclului trebuie să fie mai mică de 20 milisecunde; cea mai bună valoare este aproximativ 10 ms.

Putem modifica cu uşurinţă programul *blink* de la pagina 1 pentru a genera diferiţi factori de umplere. Tot ceea ce avem de făcut este să modificăm constantele `P` şi `DF`:

```
#define P  1250L
#define DF 30L
#define TH (P*DF/100)
```

```
//125L = 1ms
```

Valorile `P` şi `DF` de mai sus fac ca LED-ul să stea aprins numai aproximativ o treime din ciclu aprins-stins. Dacă vizualizăm pe osciloscop semnalul care controlează LED-ul, obţinem forma de undă din figura următoare:



Linia punctată din figură indică luminozitatea aparentă a LED-ului. Aceasta este aproximativ 30% din luminozitatea unui LED aprins tot timpul.

Variaţia factorului de umplere al unui semnal dreptunghiular ne permite să obţinem efectele unui semnal analogic folosind tehnici digitale. Intensitatea LED-ului ar putea fi modificată şi dacă în loc să-l alimentăm la 5V l-am alimenta de la o sursă de tensiune variabilă. Tensiunea variabilă se poate genera cu un convertor numeric analogic, dar un asemenea bloc are componente analogice, motiv pentru care este mai scump.

Tehnica variaţiei factorului de umplere se numeşte PWM – **P**ulse **W**idth **M**odulation. Pe lângă controlul luminozităţii LED-urilor, tehnica PWM este larg folosită pentru controlul turaţiei motoarelor de curent continuu.

Programul *blink* în care s-a implementat PWM-ul pentru controlul luminozităţii LED-ului se bazează pe durata de execuţie a buclei `while(1)`. Aşa cum s-a arătat în capitolul 1, timpul de execuţie al buclei `while(1)`poate varia foarte mult de la o iteraţie la alta. Din acest motiv canalele PWM se implementează cu numărătoare. Numărătorul plus blocul de generare a formei de undă implementează hardware algoritmul implementat software în programul *blink*.

În figura următoare este prezentată timerul 0 de la ATmega16 în modul fast PWM. În figură s-au reprezentat numai blocurile active în acest mod:

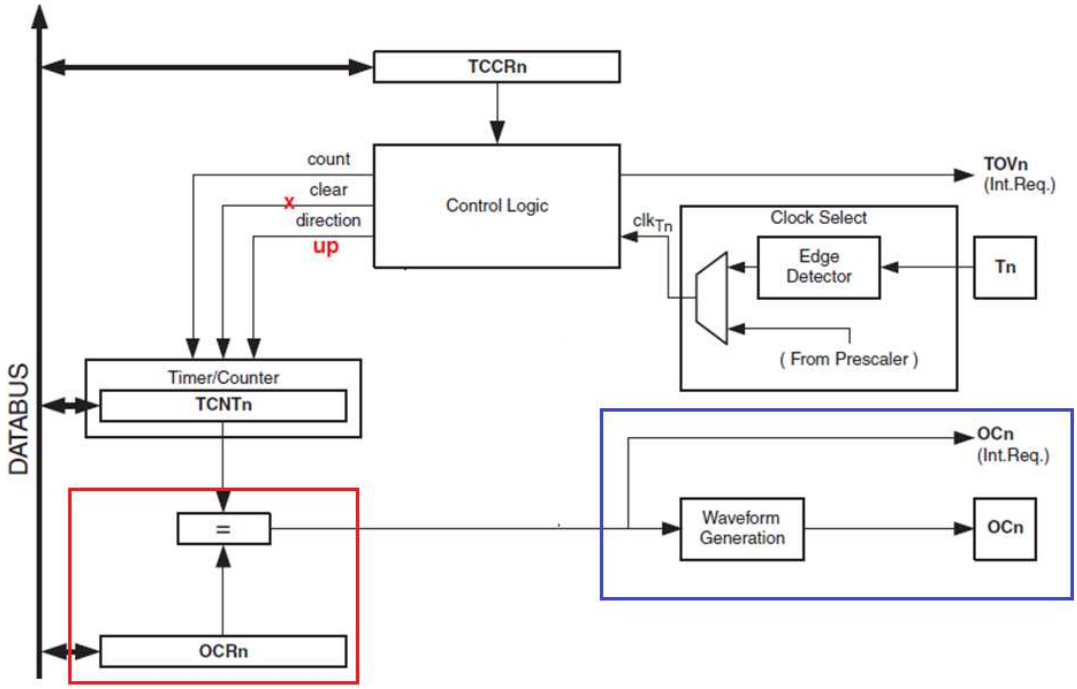


figura 8

Figura este aproape identică cu schema modului CTC din figura 7, cu excepţia legăturii dintre ieşirea comparatorului şi blocul de control.

Timerul 0 in mod fast PWM se comportă exact ca în modul normal: direcția de numărare este numai înainte, fără ca să se execute vreodată ștergerea numărătorului. Starea curentă a numărătorului se obţine din starea anterioară la care se adună 1, cu o excepție: după starea MAX (0xFF) urmează starea BOTTOM (0x00). Diferenţă dintre modul normal şi modul fast PWM o face blocul de generare a formei de undă „Waveform Genaration”. Acest bloc este marcat cu albastru în figura 8.

Blocul de generare a formei de undă controlează ieşirea Output Compare (OC0). Acest bloc este configurat în funcţie de biţii COM01 şi COM00 din TCCR0 (figura 5). Există 4 posibilităţi:

1. COM0(1:0)=00. OC0 este **deconectat**
2. COM0(1:0)=01. **Rezervat**. Nu se foloseşte.
3. COM0(1:0)=10. Mod **neinversat**: ieşirea OC0 este resetată când comparatorul detectează egalitate între starea numărătorului şi registrul OCR0 şi setată pe starea BOTTOM (0x00).
4. COM0(1:0)=11. Mod **inversat**: ieşirea OC0 este setată la egalitate şi resetată pe BOTTOM.

Pentru ca ieşirea OC0 să asculte de blocul de generare a formei de undă aceasta trebuie programată ca ieşire digitală, adică trebuie ca DDRB(3) să fie ‚1’. Suplimentar trebuie ca COM0(1:0)=”10” sau „11”.

Dacă blocul de generare este configurat în modul neinversat, ieşirea OC0 va avea timingul din figura următoare:

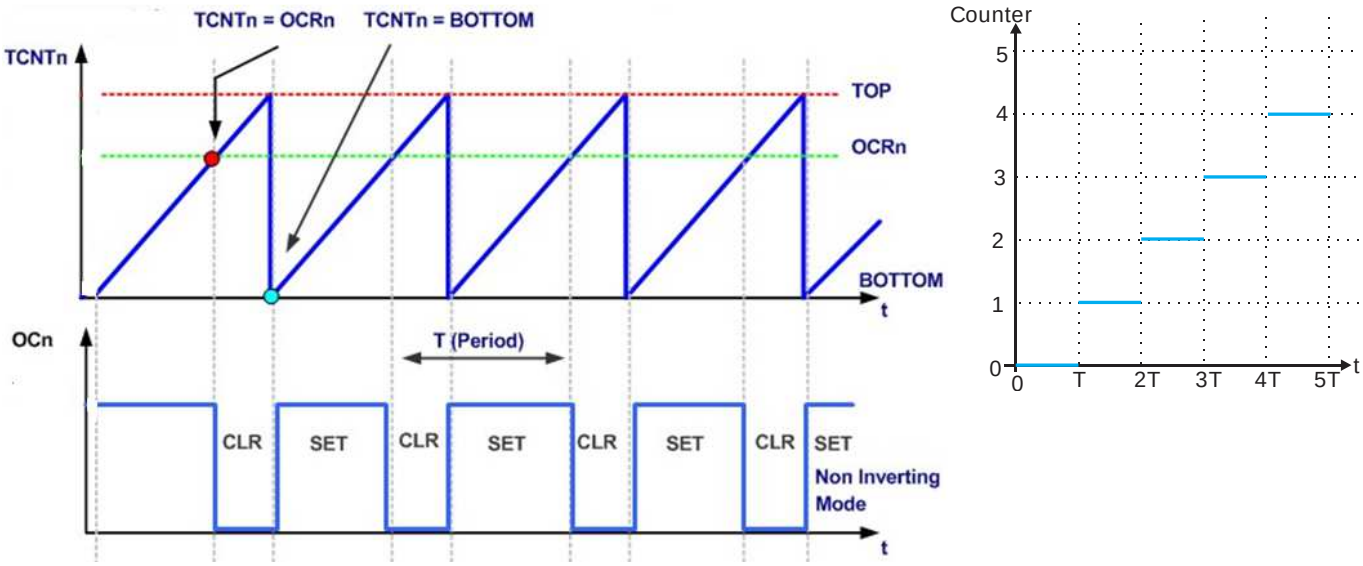


figura 9

Atenție: liniile oblice din figură nu reprezintă un semnal analogic rampă ci reprezintă starea numărătorului. În partea dreaptă este prezentată evoluția numărătorului. Starea numărătorului s-a reprezentat cu culoare bleu. La momentul t=0 starea numărătorului este 0. Această valoare nu se modifică până la apariția frontului activ al semnalului de ceas. La momentul T primul front activ al ceasului determină numărătorul să treacă în starea 1. Starea 1 nu se modifică până la următorul front de la 2T. La 2T numărătorul trece în starea 2 ș.a.m.d. În partea stângă a figurii liniile oblice sunt de fapt alcătuită din 256 de linii orizontale care nu se mai pot distinge din cauza rezoluţiei limitate.

Semnalul OCn din figura 9 este generat conform tehnicii PWM: dacă valoarea din OCR se apropie de 255 factorul de umplere va creşte către 100% iar dacă OCR scade spre 0, factorul de umplere se va duce şi el către 0%. Acest semnal poate fi folosit în exterior a controla luminozitatea unui LED sau turaţia unui motor DC sau BLDC.

În exemplul următor se va configura timerul 0 pentru a aprinde un LED exact cum o făcea programul *blink*.

În acest exemplu frecvenţă ceasului microcontrolerului este f_{CLK_CPU} **= 10 MHz**. Frecvenţa PWM trebuie să fie în plaja 50 - 200Hz (20ms - 5 ms), de preferat cât mai aproape de 100Hz. Pentru ca să parcurgă un ciclu numărătorul numără 256 de impulsuri ale ceasului de numărare CLK_{CNT}. Astfel că *N* din Rel. 3 este 256 iar Rel. 3 devine:

$$f_{CLK\_CPU} = p*N*fcycle = 256*p*\ fcycle \rightarrow\ f_{cycle} = \frac{f_{CLK\_CPU}}{256*p} \qquad \textbf{Rel. 7}$$

Valorile lui *p* pentru timerul 0 sunt 1, 8, 64, 256 şi 1024. Mai întâi vom calcula f_{cycle} pentru p=256:

$f_{cycle256} = f_{CLK\_CPU}/256/256 = 14.400.000Hz /256/256 = 219.72$ Hz

Deoarece se observă cu uşurinţă că $f_{cycle1024} = f_{cycle256}/4$ şi că $f_{cycle64} = f_{cycle256}* 4$. Vom calcula $f_{cycle256}$ şi $f_{cycle64}$ pe baza $f_{cycle1024}$:

$f_{cycle1024} = f_{cycle256}/4=219.72 /4=$**54.93Hz** $f_{cycle64} = f_{cycle256}* 4=219.72 * 4=878.90$

Cea mai apropiată valoare de 100Hz este 54.93 obţinută pentru *p*=**1024**. Această valoare este singura valoare în plaja 50 – 200Hz.

Deoarece la examen nu sunt permise mijloace electronice de calcul vom încerca aflarea lui *p* evitând împărţiri laborioase. Mai mult, cu excepţia cazurilor când se impune un anume T_{cycle}, putem aproxima pe *p* in loc să îl calculam cu exactitate. În acest exemplu se cere ca f_{cycle} să fie în plaja 50-200Hz, aşa că o eroare de 5% nu este o problemă.

Vom aproxima $f_{cycle256} = f_{CLK\_CPU}/256/256$. Mai întâi vom determina dacă f_{CLK_CPU} se divide la 3, 9, o putere a lui 10 (de obicei 1000, 1.000.000) sau o putere a lui 2 (de obicei 1024). Dacă scriem 14,4 MHz ca 14.400.000Hz se vede imediat că acest număr se divide la 100000.

Putem scrie că $14.400.000 = 144 * 100000=144*10^5$.

Cum $144=12^2$ rezulta că $14.400.000 = 144 * 10000 = 12^2\ 10^5 = (3*4)^2(2*5)^5 = 3^2\ 2^4\ 2^5\ 5^5=2^9\ 3^2\ 5^5$.

Acum împărţirea la $256=2^8$ este imediată: $(2^9\ 3^2\ 5^5)/2^8 = 2\ 3^2\ 5^5$. Astfel se evită împărțirea clasică la 256.

În continuare avem de împărţit $2\ 3^2\ 5^5$ la 256.

În acest caz putem face o aproximare: $(2\ 3^2\ 5^5)/256 = (2\ 5^3\ 3^2\ 5^2)/256 =$

$= (\cancel{250}\ 3^2\ 5^2)/\cancel{256} \cong 3^2\ 5^2=9\ 25=225$.

Se observă că rezultatul aproximat 225 este foarte apropiat de rezultatul exact 219,7. **Încă o data precizăm că aproximările se pot folosi numai dacă nu se impune un anume T_{cycle}.**

Schema de conectare a LED-ului la microcontroler este următoarea:

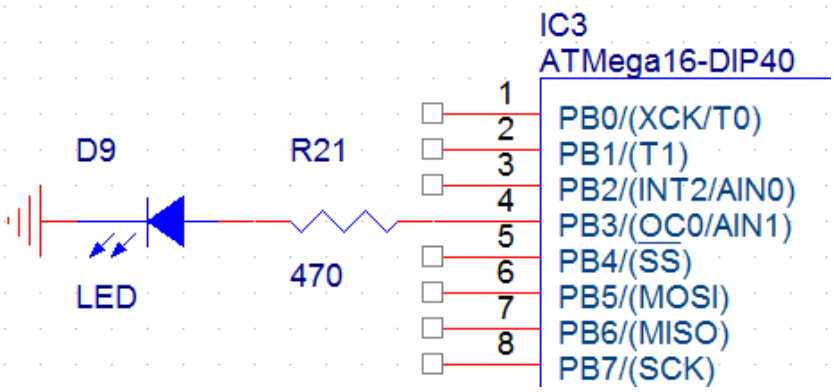


figura 10

Programul pentru *p*=1024 este următorul:

```
int main(){
   //          ┌──┬─Fast PWM
   TCCR0=0b01101101;
   //          || └┴┴─ clk   /1024
                          I/O
   //          └┴──────Set OC0 on BOTTOM, Clear on compare.

   OCR0=128; //50% Duty cycle

   setbit(DDRB,3); //OC0 pin is output. Mandatory!

   while(1){
   //……
   }//end while

}//end main
```

În exemplul anterior ieşirea OC0 controlată de numărătorul 0 în mod fast PWM a fost folosită pentru a controla intensitatea unui LED. Acelaşi mecanism poate fi folosit pentru a controla intensitatea a 7 LED-uri, adică a unui afişor 7 segmente. Schema de conectare a unui afişor 7 segmente a cărui intensitate este controlată PWM este prezentată în figura următoare:

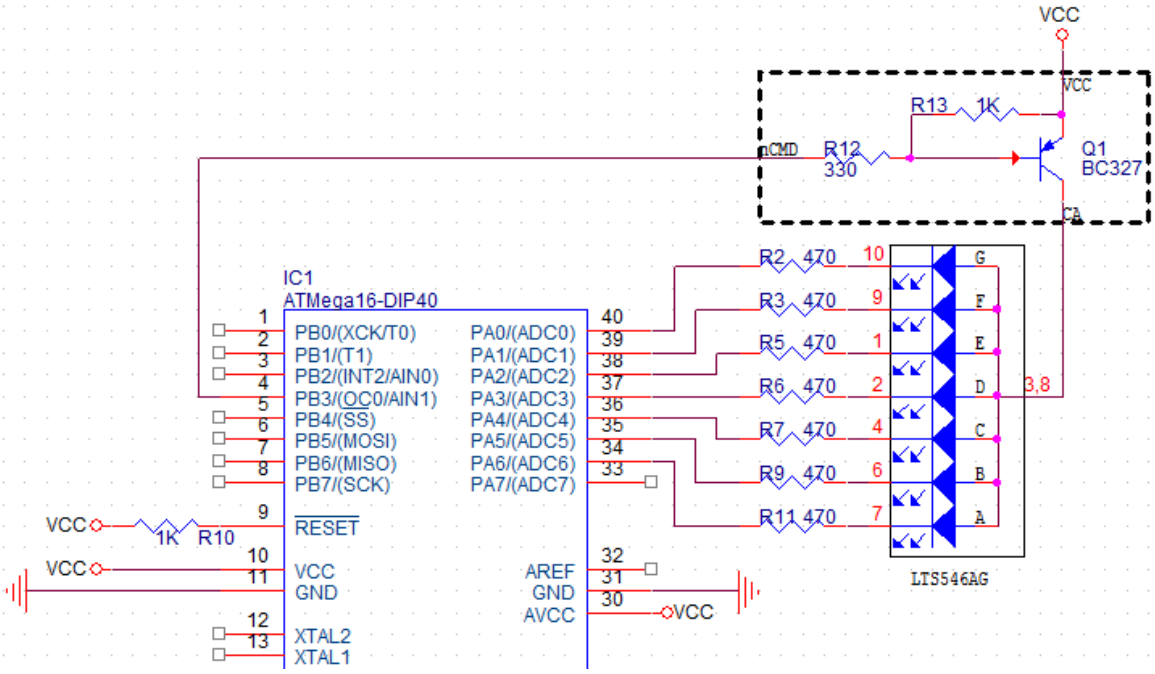


figura 11

Schema din figura 11 are la baza figura 2 din laboratorul 4. Diferenţa constă în modul de conectare a anodului comun la VCC. În figura 2 din laboratorul 4 anodul comun este conectat permanent la VCC si astfel intensitatea luminoasă a oricărui segment este fixă şi maxim posibilă. În figura 11 anodul comun se

conectează la VCC prin intermediul unui comutator electronic. Comutatorul electronic este necesar deoarece OC0 în mod ieşire poate furniza un curent maxim de 20 mA iar afişorul are nevoie de un curentul maxim de 140 mA. Comutatorul sub comanda pinului OC0 conectează sau nu anodul comun la VCC. Când OC0 = ‚0' anodul comun este conectat la VCC şi afişorul va lumina iar când OC0='1' anodul comun este deconectat şi afişorul va fi stins. Factorul de umplere al lui OC0 va determina intensitate afişorului 7 segmente.

Programul care controlează intensitatea afişorului este aproape identic cu programul care controlează intensitatea LED-ului din figura 10 cu o deosebire: LED-ul luminează când OC0='1' iar afişorul luminează când OC0='0'. Din acest motiv blocul de generare a formei de undă este configurat în mod invers, ca mai jos:

```
//         ┌┬──Fast PWM
TCCR0=0b01111101;
//         ||└┴┴─ clk   /1024
                     I/O
//         └┴──────Reset OC0 on BOTTOM, Set on compare.
```

La examen nu este necesar să proiectaţi comutatorul electronic; este suficient să desenați dreptunghiul reprezentat cu linie întreruptă în figura 11. În interiorul dreptunghiului este obligatoriu să apară pinii VCC, nCMD şi CA.

Ultimul mod de lucru al timerului 0 este modul **phase corect PWM**. Deoarece acest mod are sens numai dacă numărătorul este prevăzut cu cel puţin două canale PWM şi numai pentru o gamă redusă de aplicaţii, acest mod nu va mai fi discutat şi nu constituie subiect de examen.

3.6 Ceas digital, metoda 2

A doua metodă de implementare a ceasului digital se foloseşte când T_{cycle} nu se poate alege conform metodei descrise în capitolul 3.4, ci este impus. Este posibil ca pentru un T_{cycle} impus să nu există nici un număr $k \in N$ pentru care să fie adevărată Rel. 4, $\mathbf{T_r=kT_{cycle}}$ pe care se bazează implementarea ceasului digital. Această situaţie apare atunci când timerul se foloseşte în dublu scop: PWM şi ceas digital.

În exemplu anterior, din capitolul 3.5, timerul 0 este folosit în mod PWM. În acest mod N nu poate fi ales, valoarea sa fiind întotdeauna 256. În capitolul precedent s-a calculat $f_{cycle1024}$:

$$f_{cycle1024} = 54{,}93\ Hz \quad \rightarrow \quad T_{cycle1024} = \frac{1}{f_{cycle1024}} = 0.01820(4)s$$

Pentru un T_r specific ceasului digital (o secundă sau jumătate de secundă sau o zecime de secundă) nu există un număr $k \in N$ astfel încât să fie adevărată Rel. 4, $\mathbf{T_r=kT_{cycle}}$. Pentru un Tr=1s vom obţine k=T_r/T_{cycle}=1s/ T_{cycle}= f_{cycle}=54,93. Evident, k nu este întreg!

Metoda care va fi prezentată în continuare se mai poate folosi şi atunci când din calculele făcute în capitolul 3.4 pentru ceas digital rezultă o valoare a lui k este prea mare. Conform celor discutate anterior un k mare înseamnă o valoare mică pentru T_{cycle} iar o valoare mică pentru T_{cycle} înseamnă un timp mic pentru bucla `while(1)`. Indiferent de motivul pentru care se foloseşte metoda, pentru ca să o putem aplica condiţia iniţială $\mathbf{T_r \geq T_{cycle}}$ din capitolul 3.4 trebuie îndeplinită.

Ideea metodei pe care o vom folosi este uşor de înțeles prin intermediul următoarei analogii. Să presupunem că avem un dispozitiv de măsurare a timpului care emite un semnal luminos sau sonor din 25 în 25 de secunde. În afară de acest semnal dispozitivul nu are nici un alt sistem de semnalizare a timpului curs. Se cere să măsuram timpul scurs în minute. În acest caz T_{cycle} = 25s iar T_r=60s. Nu există nici un număr natural k astfel încât 60=k 25.

Soluția acestei probleme este ușor de imaginat. Știm că de fiecare dată când apare semnalul de la dispozitiv au trecut T_{cycle} =25 de secunde. La prima activare au trecut 25 de secunde, la a doua activare au trecut 50, la a treia au trecut 75, etc. În general la a *i*-a activare a semnalului s-au scurs *i**25 secunde.

Numărul de minute care s-au scurs este câtul împărțirii secundelor scurse la T_r=60s. În continuare câtul împărțirii a nouă numere naturale $a,\ b\ \in N,\ b \neq 0$ se notează cu $q = \left\lfloor \frac{a}{b} \right\rfloor$ sau cu $q = \lfloor a/b \rfloor$. În continuare semnalul de la dispozitiv se abreviază SD. Evidența minutelor scurse în funcție de activările semnalului este următoarea:

Tabelul 1

Număr apariție SD: i	Secunde scurse: i * 25	Minute scurse: q=$\lfloor (i*25)/60 \rfloor$	Rest: r	Eroare suplimentară a metodei 2
0	**0**	**0\60 = 0**	**0**	**0**
1	0 + 25 = 25	25\60 = 0	25	0
2	25 + 25 = 50	50\60 = 0	50	0
3	**50 + 25 = 75**	**75\60 = 1**	**15**	**15**
4	15 + 25 = 100	100\60 = 1	40	0
5	**40 + 25 = 125**	**125\60 = 2**	**5**	**5**
6	125 + 25 = 150	150\60 = 2	30	0
7	150 + 25 = 175	175\60 = 2	55	0
8	**175 + 25 = 200**	**200\60 = 3**	**20**	**20**
9	200 + 25 = 225	225\60 = 3	45	0
10	**225 + 25 = 250**	**250\60 = 4**	**10**	**10**
11	**250 + 25 = 275**	**275\60 = 4**	35	0
...				

Formula de calcul a minutelor scurse $q = \lfloor (i * 25)/60 \rfloor$ este foarte simplă dar implementarea în C (sau alt limbaj de programare) ridică o problemă. Pentru a înțelege problema vom considera a unsprezecea apariție a semnalului. Această apariție este marcată cu roșu in tabelul de mai sus. Dacă pentru evidență secundelor scurse se alocă o variabilă de tip *unsigned char*, atunci la a 11-a apariție apare depășire.

O variabilă de tip *unsigned char* se reprezintă pe 8 biți, așa că cea mai mare valoare care se poate reprezenta este 255. Pe 8 biți **250 + 25 = 19**, nu 275 așa cum ar trebui. Se poate argumenta că depășirea a apărut pentru că 8 biți sunt prea puțin. Într-adevăr, dar depășirea va apare indiferent de numărul de biți pe care se reprezintă secundele, doar că va apare mai târziu. De exemplu, pentru T_{cycle} = 0,01s și variabilă *int* pe 32 de biți depășirea apare după aproximativ 500 de zile. Deși 500 de zile pare foarte mult, în realitate nu este așa. Există sisteme de control în mediu industrial și energetică care funcționează fără oprire ani de zile. De asemenea serverele pot funcționa fără întrerupere ani de zile. Imaginați-vă ce s-ar întâmpla dacă într-un sistem de control dintr-o centrală nucleară s-ar reseta timpul!

Revenind la exemplul anterior, nu am făcut altceva decât sa convertim timpul măsurat de un ceas cu rezoluția de 25 de secunde în timp măsurat în unități de 60 de secunde (adică minute). În exemplu T_{cycle} = 25s iar T_r=60s. Timpul măsurat în minute s-a calculat cu formula $q = \lfloor (i * 25)/60 \rfloor$.

În general, timpul exprimat în noua unitate de timp se calculează cu formula generală

$$q_i = \left\lfloor \frac{i\ T_{cycle}}{T_r} \right\rfloor$$

unde q_i este câtul corespunzător unui timp de scurs de $i\ T_{cycle}$.

Pentru a elimina produsul $i\ T_{cycle}$ care poate produce depășire vom aplica teorema împărțirii cu rest pentru doi timpi succesivi exprimați în unități T_{cycle}: $i\ T_{cycle}$ și $(i+1)\ T_{cycle}$.

Mai întâi aplicăm teorema împărțirii cu rest pentru deîmpărțitul $i\ T_{cycle}$ și împărțitorul T_r:

$$i\ T_{cycle} = q_i\ T_r + r_i$$

Apoi pentru deîmpărţitul $(i+1)$ T_{cycle} şi împărţitorul T_r:

$$(i+1)T_{cycle} = q_{i+1}\,T_r + r_{i+1}$$

Facem diferenţă celor două egalităţi şi astfel **eliminăm produsul *i* T_{cycle}**:

$$(i+1)T_{cycle} - i\,T_{cycle} = q_{i+1}\,T_r + r_{i+1} - q_i\,T_r - r_i$$

$$T_{cycle} = (q_{i+1} - q_i)\,T_r + r_{i+1} - r_i$$

Din egalitatea de mai sus calculăm restul la a i+1 –a apariţie a lui SD:

$$r_{i+1} = r_i + T_{cycle} - (q_{i+1} - q_i)\;T_r$$

Deoarece $\mathbf{T_r \geq T_{cycle}}$, atunci ori $q_{i+1} = q_i$, ori cu $q_{i+1} = q_i+1$. Acest fapt se observă şi în Tabelul 1: nu există nici o pereche de linii consecutive pentru care diferenţa între q-urile corespunzătoare să fie mai mare ca unu. Din două variante pentru q_{i+1} rezultă două variante pentru r_{i+1}:

$$r_{i+1} = r_i + T_{cycle} - T_r \quad dacă \quad q_{i+1} = q_i + 1$$

$$r_{i+1} = r_i + T_{cycle} \qquad dacă \quad q_{i+1} = q_i$$

În cazul de faţă, al conversiei timpului din unităţi T_{cycle} în unităţi T_r, atât deîmpărţitul cât şi împărţitorul sunt pozitive, ceea ce face ca restul să fie pozitiv şi mai mic ca împărţitorul. Într-o prima aproximaţie putem calcula ambele variante de rest şi apoi să eliminăm restul negativ sau restul mai mare sau egal cu împărţitorul. Teorema împărţirii cu rest garantează unicitatea restului, adică unul din resturi va fi cel corect.

Pe baza raţionamentului şi observaţiilor de mai sus construim Tabelul 2 în care calcularea timpului în unităţi T_r se face pe baza restului, nu a produsului *i* T_{cycle}. În tabel se calculează ambele variante de rest şi se selectează varianta corectă. Varianta corectă este marcată cu ☑ iar cea rejectată, cu ☒. În tabel apare şi coloana *i* T_{cycle} = *i* * 25 dar valorile din această coloană nu se folosesc la calcularea timpului în unităţi T_r=60 ci doar la verificarea corectitudinii calculelor.

Tabelul 2

Număr apariţie SD: i	**Secunde scurse: i * 25**	$r_{i+1} = r_i + T_{cycle} - T_r$ sau $r_{i+1} = r_i + T_{cycle}$	**Minute scurse: q**	**Rest: r**
0	**0**	**0**	**0**	**0**
1	0 + 25 = 25	r = 0+25-60 = -35☒ **r = 0+25 = 25☑**	q = 0+1 = 1☒ **q = 0 ☑**	25
2	25 + 25 = 50	r = 25+25-60 = -10☒ **r = 25+25 = 50☑**	q = 0+1 = 1☒ **q = 0☑**	50
3	**50 + 25 = 75**	**r = 50+25-60 = 15☑** r = 50+25 = 75☒	**q = 0+1 = 1☑** q = 0☒	**15**
4	15 + 25 = 100	r = 15+25-60 = -20☒ **r = 15+25 = 40☑**	q = 1+1 = 2☒ **q = 1☑**	40
5	**40 + 25 = 125**	**r = 40+25-60 = 5☑** r = 40+25 = 65☒	**q = 1+1 = 2☑** q = 1☒	**5**
6	125 + 25 = 150	r = 5+25-60 = -30☒ **r = 5+25 = 30☑**	q = 2+1 = 3☒ **q = 2☑**	30
7	150 + 25 = 175	r = 30+25-60 = -5☒ **r = 30+25 = 55☑**	q = 2+1 = 3☒ **q = 2☑**	55
8	**175 + 25 = 200**	**r = 55+25-60 = 20☑** r = 55+25 = 80☒	**q = 2+1 = 3☑** q = 2☒	**20**
...				

Calculele de mai sus sunt foarte simple dar implică variabile de tip flotant. Multe calculele cu variabile *float* sau *double* pot produc erori datorită imposibilităţii reprezentării anumitor numere cu număr

finit de cifre. De exemplu, $T_{cycle1024} = 0.01820(4)s$ nu poate fi reprezentat exact decât cu număr infinit de cifre. Putem elimina calculele în virgulă mobilă dacă exprimăm toate cantitățile ca multipli de T_{CLK_CPU} şi apoi simplificăm prin T_{CLK_CPU}. În acest sens vom face trei observaţii:

1. Expresia lui $T_{cycle} = p * N * T_{CLK\_CPU}$ din Rel. 2 ne spune că T_{cycle} este un multiplu a lui T_{CLK_CPU}.
2. Dacă şi T_r este multiplu de T_{CLK_CPU}, atunci putem scrie T_r = PPTR T_{CLK_CPU}. PPTR înseamnă **P**ulsuri T_{CLK_CPU} **p**er **Tr**. Dacă T_r nu este un multiplu de T_{CLK_CPU}, atunci eliminarea calculelor în virgulă mobilă este imposibilă.
3. Dacă atât T_{cycle} cât şi T_r din relaţia $i\,T_{cycle} = q_i\,T_r + r_i$ sunt multipli de T_{CLK_CPU} se poate demonstra foarte uşor că şi r_i este multiplu de T_{CLK_CPU}: $r_i = r_i' * T_{CLK\_CPU}$.

Dacă exprimăm toate cantităţile în multipli de T_{CLK_CPU} toate calculele în virgulă mobilă pot fi transformate în calcule cu numere întregi. Cele două forme ale restului se transformă după cum urmează:

$$r_{i+1} = r_i + T_{cycle} \quad \rightarrow \quad r'_{i+1}T_{CLK\_CPU} = r'_i T_{CLK\_CPU} + pN\,T_{CLK\_CPU} \xrightarrow{/T_{CLK\_CPU}}$$

$$r'_{i+1} = r'_i + pN \qquad \text{varianta 1}$$

$$r_{i+1} = r_i + T_{cycle} - T_r \quad \rightarrow \quad r'_{i+1}T_{CLK\_CPU} = r'_i T_{CLK\_CPU} + p\,N\,T_{CLK\_CPU} - PTTR\,T_{CLK\_CPU} \xrightarrow{/T_{CLK\_CPU}}$$

$$r'_{i+1} = r'_i + pN - PTTR \qquad \text{varianta2}$$

În continuare vom rafina procesul de calcul al restului r'_{i+1}. Mai întâi observăm că suma $r'_i + pN$ apare în ambele variante de calcul ale lui r'_{i+1}. Restul r'_{i+1} este chiar această sumă în varianta 1. Pentru varianta 2 din sumă trebuie să scădem PTTR. Dacă analizăm Tabelul 2 se observă că restul se calculează conform variantei 2 atunci când acesta este pozitiv. Condiția $r'_{i+1} \geq 0$ înseamnă $r'_i + pN - PTTR \geq 0 \quad \rightarrow \quad r'_i + pN \geq PTTR$.

Pe baza acestei observaţii putem rescrie modul de calcul al restului:

$s = r'_i + pN$
$if\ s \geq PTTR\ \{$
    $r'_{i+1} = s - PTTR$
    $q{+}{+}$
}
else
    $r'_{i+1} = s$

În rezumat, la terminarea unui ciclu de numărare acumulăm *p**N pulsuri şi comparăm valoarea acumulată cu PTTR. Dacă am acumulat mai multe pulsuri decât PTTR scădem din numărul de pulsuri acumulate PTTR pulsuri şi păstrăm diferenţa pentru următorul ciclu.

La începutul acestui capitol s-a precizat că metoda 2 se foloseşte atunci când un timer se configurează unui prim scop, de exemplu PWM, dar se doreşte utilizarea lui şi pentru un al doilea scop, de exemplu ceas digital. În capitolul precedent, Modul fast PWM – fast Pulse Width Modulation, timerul 0 a fost configurat în mod fast PWM cu *p*=1024. Ceasul extern CLK_CPU are frecvenţa de 14,4 MHz. Pentru implementarea ceasului digital alegem T_r=1s. In acest caz avem

$$T_r = PTTR\ T_{CLK\_CPU} \rightarrow PTTR = \frac{T_r}{T_{CLK\_CPU}} = T_r\,f_{CLK\_CPU} = 1s\ 14.400.000Hz = 14.400.000$$

Implementarea in C este următoarea :

```
#define p 1024
#define PTTR 14400000UL

int main(){
   unsigned long r = 0UL; //restul
   unsigned long sec = 0UL; //secunde

   //        ┌──┬─Fast PWM
   TCCR0=0b01101101;
   //        || └┴┴ clk   /1024
   //        └┴─────Set OC0 on BOTTOM, Clear on compare.
                      I/O

   OCR0=128; //50% Duty cycle

   setbit(DDRB,3); //OC0 pin is output. Mandatory!

   while(1){
      if(TIFR & 1<<TOV0){  // TOV0 este definit in avr/io.h
         TIFR |= 1<<TOV0;  // clear TOV0

         r += p * 256UL;   //p*N
         if(r >= PTTR){
            r -= PTTR;
            sec++;
            //update time, display time
         }
      }
   }//end while
}//end main
```

Există însă şi un dezavantaj al metodei 2 şi anume o scădere suplimentară a acurateţei. Dacă în exemplul anterior în care dispuneam de dispozitiv care emite un semnal din 25 în 25 de secunde vrem să afişăm minutele, din Tabelul 1 constatăm că minutul 1 este afişat pentru prima dată când timpul scurs este 1minut şi 15 secunde. Eroarea suplimentară este în acest caz de 15 secunde. Erorile suplimentare sunt trecute în Tabelul 1 in coloana „Eroarea suplimentară a metodei 2”.

Creşterea suplimentară a erorii este de maxim un ciclu de numărare T_{cycle}. Această valoare trebuie adăugată la eroarea calculată conform relaţiei Rel. 5.

$$err_{poll\_metoda1} = T_{cycle} + T_{max\_while(1)} + \Delta_{min} \qquad \textbf{Rel. 8}$$

$$err_{poll\_metoda2} = err_{poll\_metoda1} + T_{cycle} = 2T_{cycle} + T_{max\_while(1)} + \Delta_{min}$$

În exemplul anterior T_{cycle} este 18.2ms. O imprecizie suplimentară de 18,2 ms este perfect acceptabilă pentru cuptoare cu microunde şi maşini de spălat dar este inacceptabilă pentru un cronometru sportiv.

4. Timer/Counter-ul 2

Numărătorul 2 seamănă foarte mult cu numărătorul 0. Diferenţă constă în modul în care este utilizat ceasul extern de numărare. Blocul în care se procesează acest semnal este prescalerul. În figura 12a este reluat prescalerul timerului 0 iar în secţiunea b) a figurii este prezentat prescalerul timerului 2. În ambele figuri ceasul extern şi a fost marcat cu roşu iar cel intern cu albastru.

La timerul 0 ceasul extern T0 este procesat în blocul numit „Synchronization” în figura 12a. Acest bloc foloseşte ceasul procesorului $clk_{I/O}$ pentru a analiza ceasul extern T0 şi generează la ieşire un puls fie pentru frontul ridicător al lui T0, fie pentru cel coborâtor, în funcţie de programare. Din figură se observă că semnalul rezultat din ceasul extern T0 devine ceasul numărătorului 0 fără a mai putea fi divizat de prescaler.

La timerul 2 din figura 12b fie semnalul extern TOSC1, fie $clk_{I/O}$ sunt mai întâi divizate şi apoi devin ceas pentru timerul 2. Adică ceasul extern TOSC1 se poate diviza la fel ca şi $clk_{I/O}$.

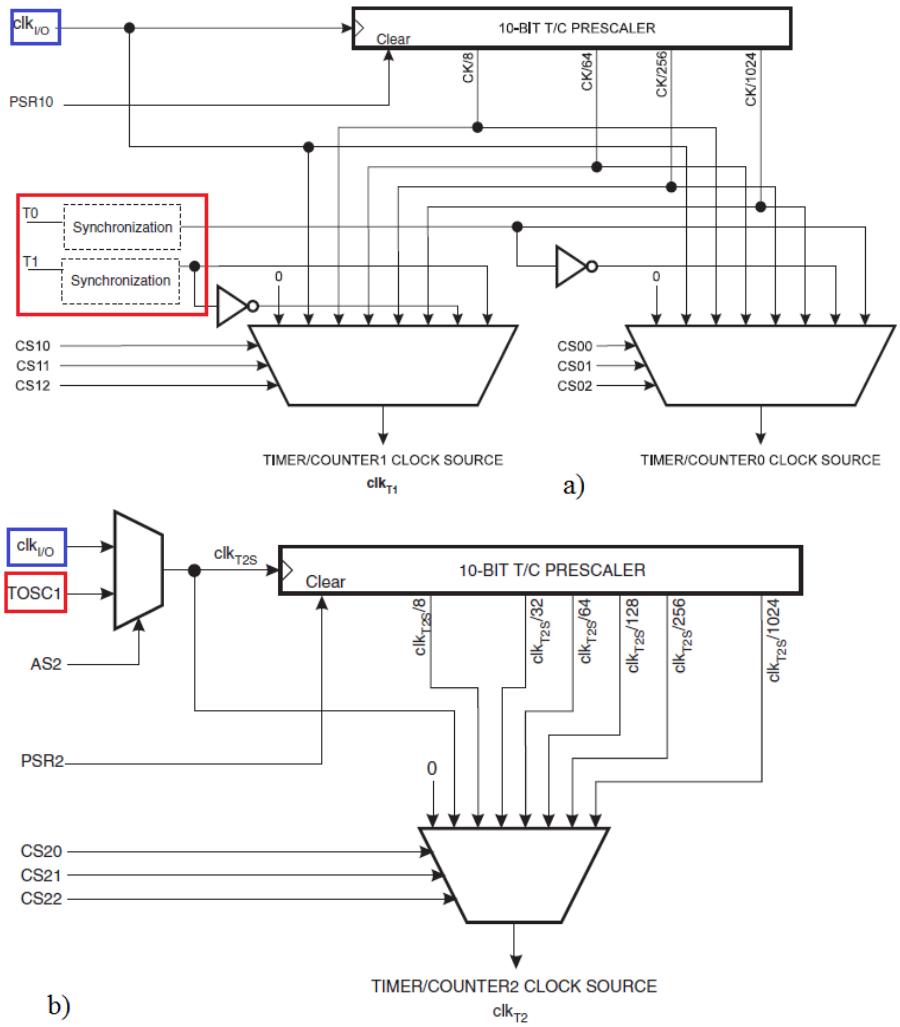


figura 12

Atenţie: oscilatorul care generează ceasul TOSC1 este optimizat pentru cristale cu frecvenţa de 32,768 kHz. Folosirea altei frecvenţe nu este recomandată.

Dacă ambele timere folosesc doar ceasul intern $clk_{I/O}$ apare o diferenţă minoră în ceea ce priveşte factorii de divizate. La ambele timere factorii de divizare sunt 1, 8, 64, 256 şi 1024 dar la timerul 2 apar suplimentar şi factorii 32 şi 128.

În cazul în care timerul 2 foloseşte semnalul extern de numărare se spune că acesta funcţionează în mod asincron. Deşi este relativ simplu, modul asincron al timerul 2 nu constituie subiect de examen şi nici o problemă nu va necesita acest mod pentru rezolvare.

Registrele timerului 2 (TCCR2 şi OCR2) au exact aceeaşi structură ca registrele TCCR0 şi OCR0 asociate timerului 0. Semnificaţia biţilor este identică în aceste registre cu excepţia celor trei biţi CS care controlează prescalerul. Modurile de lucru normal, CTC, fast PWM şi phase correct PWN sunt şi ele identice la cele două timere. Indicatorii timerului 2 sunt TOV2 şi OCF2. Aceştia au aceeaşi semnificaţie ca TOV0 şi OCF0 şi se pot accesa prin intermediul biţilor 7 şi 6 din TIFR (figura 6).