1. INTRODUCERE ÎN ECHIPAMENTELE DE LABORATOR

1.1 OBIECTIVE

În continuare, sunt prezentate echipamentele ce vor fi utilizate în cadrul lucrărilor de laborator:

- Sursa de tensiune programabilă **HAMEG HM8040-3**;
- Sursa de tensiune programabilă **HAMEG HM7042-5**;
- Generatorul de funcții programabil **HAMEG HM 8030-6**;
- Multimetru digital programabil **HAMEG HM8012**;
- Frecvențmetru numeric 1.6GHz **HAMEG HM8021-4**;
- Generatorul de forme de undă **SDG2000X**;
- Sarcina electronică programabilă **SDL1000X**;
- Multimetrul digital **SDM3065X**;
- Osciloscopul digital programabil **SDS2000X Plus**;
- Sursa de alimentare programabilă **SPD3303X**;
- Sursa de curent alternativ programabilă **ITECH IT7324**;
- Sursa de alimentare AC/DC programabilă **ITECH IT7622**;
- Sursa de alimentare DC de mare putere **ITECH IT6522C**.

1.2 SURSA DE TENSIUNE PROGRAMABILĂ HAMEG HM8040-3

Definiție

O sursă de tensiune este un dispozitiv care furnizează energie electrică unui circuit sau consumator, menținând o anumită diferență de potențial (tensiune) între bornele sale, indiferent de curentul care circulă prin circuit.

O sursă de tensiune stabilizată generează la ieșire o **tensiune constantă**, independentă de eventualele fluctuații ale tensiunii de alimentare, a sarcinii alimentate sau a temperaturii. Sursa de tensiune se folosește pentru **alimentarea circuitelor** studiate în cadrul laboratorului.

Caracteristicile sursei **HM8040-3** sunt:

- 2 surse de tensiune de ieșire reglabilă între **0** și **20 V** la **0.5 A** și o sursă fixa **5 V** la **1 A**;
- Rezoluție afișată **0.1 V / 1 mA**;
- Posibilitate de conectare în paralel sau serie;
- Buton pentru activarea/dezactivarea simultană a tuturor canalelor;
- Limitare ajustabilă pentru curent și siguranță electronică.

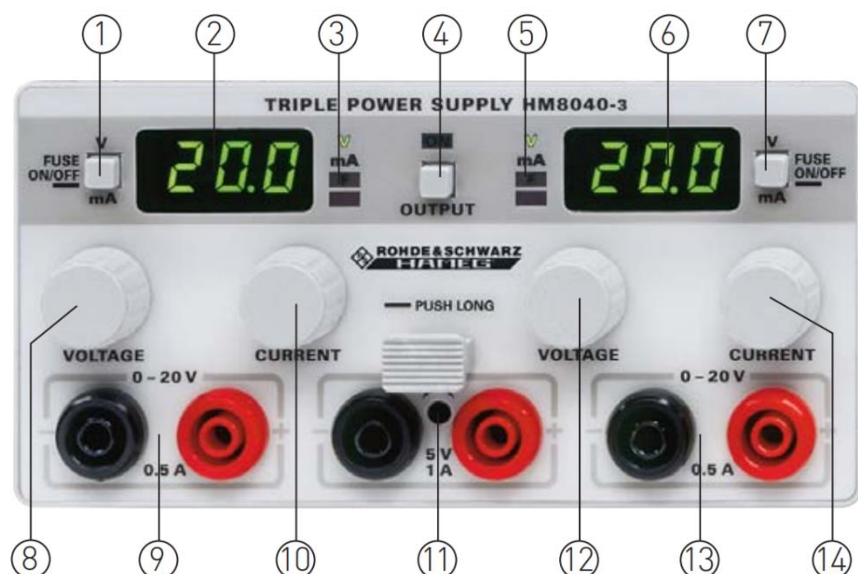


Figura 1.1 - Panoul frontal HM8040-3

1.2.1 COMENZI DE FUNCȚIONARE

- **(1) și (7) – Buton V / mA / Siguranță electronică**
  Butoanele **(1)** (**afișaj stânga**) și **(7)** (**afișaj dreapta**) sunt butoane cu ajutorul cărora se poate selecta afișarea **tensiunii** sau **curentului**, respectiv pentru **activarea siguranței electronice** individual pentru fiecare parte. Curentul este indicat cu o rezoluție de **1 mA**, iar tensiunea este afișată cu o rezoluție de **0.1 V**. Schimbarea între afișarea curentului și tensiunii se face printr-o apăsare scurtă a butonului ce are ca efect schimbarea indicatorilor **(3)** și **(5)**, iar pentru o apăsare lungă activează **siguranța electronică**, semnalată prin aprinderea indicatorului **F** din indicatorii **(3)** și **(5)**.

- **(2) și (6) – Afișaj tensiune / curent**
  Cele 2 afișaje de 3 digiți oferă afișarea selectabilă a **tensiunii de ieșire** sau a **curentului de ieșire**. Afișajul din **stânga** indică **tensiunea** sau **curentul** pentru terminalele de **ieșire** din partea **stângă (9)**, iar afișajul din partea **dreaptă** indică **parametrii** pentru terminalele de **ieșire** din partea **dreaptă (13)**.

- **(3) și (5) – LED**
  - **V – LED martor** pentru a marca valoarea afișată ca fiind **voltajul** selectat pentru ieșirea corespunzătoare în **volți**;
  - **mA – LED martor** pentru a marca valoarea afișată ca fiind **curentul** consumat de ieșirea corespunzătoare în **miliamperi**;
  - **F – LED martor** ce marchează **activarea siguranței electronice**; este obligatoriu folosirea **siguranței electronice** în timpul folosirii aparatului;
  - **I_{max} – LED martor** ce marchează **depășirea limitei impuse** pentru **curent**; dacă siguranța electronică este **activată**, ieșirea sursei de tensiune se va **deconecta automat** când se detectează **depășirea limitei de curent**; în cazul în care siguranța electronică **nu este activată** și curentul de ieșire **depășește** limita setată, **sursa de tensiune** se transformă în **sursă de curent** și limitează curentul de ieșire la valoare setată.

- **(4) – Activare / Dezactivare ieșire sursă**
  Comanda ieșirii **DC** activează simultan toate cele 3 ieșiri **DC** (buton apăsat la **HM8040-3**). Deasupra butonului este prezent un **LED martor** (marcat cu **ON**) pentru **starea ieșirii**. Afișajele de tensiune vor indica **tensiunea de ieșire**, chiar și atunci când LED-ul pentru ieșire indică faptul că ieșirea este deconectată.

- **(8) și (12) - Ajustare tensiune**
  Butoanele rotative pentru ajustarea **tensiunii** sunt folosite pentru **varierea tensiunii** în domeniul **0-20 V**. Butonul rotativ **(8)** din **stânga** setează sursa din **stânga**, iar butonul rotativ **(12)** din **dreapta** setează sursa din **dreapta**. Butoanele au **blocare mecanică** la ieșirea din domeniul tensiunii.

- **(9) și (13) - Ieșire 0 – 20 V**
  Terminalele de **ieșire** pentru sursele **0-20V** constau din **2 mufe banană mamă** la care se pot conecta **fire** sau **mufe banană tată**. Circuitul electronic asigură protecție împotriva scurtcircuitului.

- **(10) și (14) – Ajustare limită curent**
  Butoanele rotative pentru ajustarea **limitării de curent** pentru **ieșirea** din partea **stângă (10)** și **ieșirea** din partea **dreaptă (14)**. Domeniul de reglare este între **0-0.5 A**, valoarea curentului crescând la rotirea în sensul ceasornicului (**clockwise**). Butoanele au **blocare mecanică** la ieșirea din domeniul tensiunii. Atunci când butonul este rotit în sensul invers ceasornicului (**counter-clockwise**) spre poziția maximă, valoarea curentului setat scade spre **0**. Rotind la maxim spre valoarea de **0** a curentului, se **activează martorul** pentru depășirea limitei impuse de curent, chiar dacă ieșirea **nu este activată**.

- **(11) – Ieșire 5 V**
  Terminalele de **ieșire** pentru sursa de tensiune de **+5 V** constau din **2 mufe banană mamă** de **4 mm** la care se pot conecta **fire** sau **mufe banană tată**. Circuitul electronic asigură **protecție** împotriva **scurtcircuitului**. Un orificiu de acces (aflat **deasupra**, între cele două terminale de **5 V**) permite un reglaj **fin** între **4.5 V-5.5 V**.

Notă: Pentru cele trei grupuri de mufe banană (**9, 11, 13**), ieșirea de culoare **neagră** reprezintă **semnalul de masă al ieșirii**, iar cea de culoare **roșie ieșirea voltajului pozitiv**.

1.2.2 MODUL DE UTILIZARE

Reglarea aparatului pentru folosirea în laborator se face activând **modulul** (apăsând **butonul roșu** din mijlocul grupului de două module). După terminarea inițializării interfeței de afișare, se activează **siguranța** printr-o apăsare **lungă** pe **(1)** sau **(7)**, în funcție de partea modulului utilizată, până la aprinderea martorului din **(3)**, respectiv **(5)**, în cazul în care respectivii martori nu sunt activi.

Inițial, după deschidere, sursa pornește cu **toți martorii aprinși**, **voltajul** și **curentul** putând fi modificați cu **(8)**, **(10)**, **(12)**, **(14)** chiar dacă sursa este **închisă**. După ce este activată **siguranța**, se modifică **voltajul** și **curentul** la valorile dorite. Apoi, se conectează la conectorul **negru** din **(9)**, **(11)**, sau **(13)** (în funcție de partea modului utilizată) mufa corespunzătoare **mesei sistemului alimentat**, iar la cel **roșu** se conectează la **voltajul pozitiv de alimentare** a plăcii. După conectarea plăcii, se **verifică** din nou **voltajul de alimentare**, **poziția butonului** rotativ pentru **curent** și faptul că **martorul** pentru siguranța electronică **este aprins**. Apoi se apasă butonul **(4)** de activarea ieșirilor.

1.3 SURSA DE TENSIUNE PROGRAMABILĂ HAMEG HM7042-5

Hameg HM7042-5 este o **sursă de tensiune triplă**, programabilă cu următoarele caracteristici:

- 3 surse de tensiune de ieșire independente programabile: 2 cu **0-32 V**, **2A** și una cu **2.7-5.5 V**, **5A**;
- Rezoluție afișată: **10 mV / 1 mA** pe canalul **1** și **3**; **10 mV / 10 mA** pe canalul **2**;
- Posibilitate de conectare în **paralel** (până la **9A**) sau **serie** (până la **69.5 V**);
- Limitare ajustabilă pentru curent și siguranță electronică.

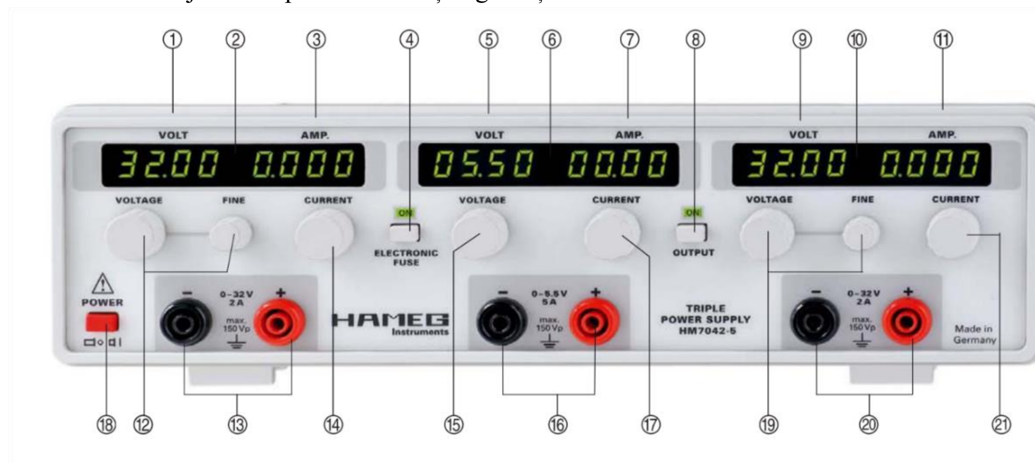


Figura 1.2 - Panoul frontal HM7042-5

1.3.1 COMENZI DE FUNCȚIONARE

- **(1), (5) și (9) – Afișaj tensiune**
  Cele 3 afișaje de 4 digiți oferă afișarea selectabilă a **tensiunii de ieșire**.

- **(2), (6) și (10) – LED**
  Martor ce indică atingerea **limitei de curent**.

- **(3), (7) și (11) – Afișaj curent**
  Cele 3 afișaje de 4 digiți oferă afișarea **curentului de ieșire**.

- **(4) – Siguranță electronică**
  Buton pentru **activarea siguranței electronice**, având un **LED martor** prezent deasupra.

- **(8) – Activare / Dezactivare ieșire**
  Activează / dezactivează simultan **toate** cele 3 ieșiri **DC**, starea fiind indicată de **LED-ul adiacent**.

- **(12) și (19) – Ajustarea tensiunii**
  Butoane pentru **ajustarea** grosieră și fină a **tensiunii** în domeniul **0-32 V**
- **(13) și (20) – 0-32V / 2A – Ieșiri, conectori de 4 mm**

- **(14), (17) și (21) – Limită curent**
  Butoanele rotative pentru **ajustarea limitării de curent** pentru **ieșirea** din partea **stângă (14)**, ieșirea **centrală (17)** și ieșirea din partea **dreaptă (21)**. Comportamentul acestei funcții cât și a siguranței electronice este asemenea celui de la sursa **HM8040-3**.

- **(15) – Ajustare tensiune**
  Butoanele rotative pentru **ajustarea tensiunii** sunt folosite pentru varierea tensiunii între **0-5.5 V**.

- **(16) – 0-5.5V / 5A – Ieșiri, conectori banană de 4 mm**

- **(18) – ON/OFF – Buton pentru deschiderea / închiderea sursei de tensiune**

1.3.2 MODUL DE UTILIZARE

După pornire, sursa de alimentare **nu va avea** ultimele setări făcute înainte de deconectarea ei, butoanele pentru reglaj putând fi **modificate** cu **sursa deconectată**. Pentru **protejarea** componentelor alimentate de la sursă, este **obligatorie activarea siguranței electronice**. Utilizând butoanele **(14)**, **(17)**, **(21)** curentul maxim (**I_{max}**) poate fi setat pentru **fiecare** din cele 3 canale. Limitarea curentului pe un canal **nu va influența** curentul pe celelalte, dar **activarea protecției** la supracurent va **dezactiva** ieșirile în caz de **depășire** a limitei setate. În caz de **atingere a limitei curentului**, se aprinde **LED-ul martor** corespunzător canalului utilizat ((**2**), **(6)** sau **(10)**).

1.4 GENERATORUL DE FUNCȚII PROGRAMABIL HAMEG HM8030-6

Definiție

Generatorul de funcții este un aparat electronic ce furnizează semnale variabile de diferite forme (sinus, dreptunghi, triunghi, impuls, etc.), permițând modificarea în funcție de anumiți parametri: amplitudine, frecvență, factor de umplere, formă. Generatorul se utilizează la aplicarea de semnale variabile în circuitele electronice, ce se studiază experimental.

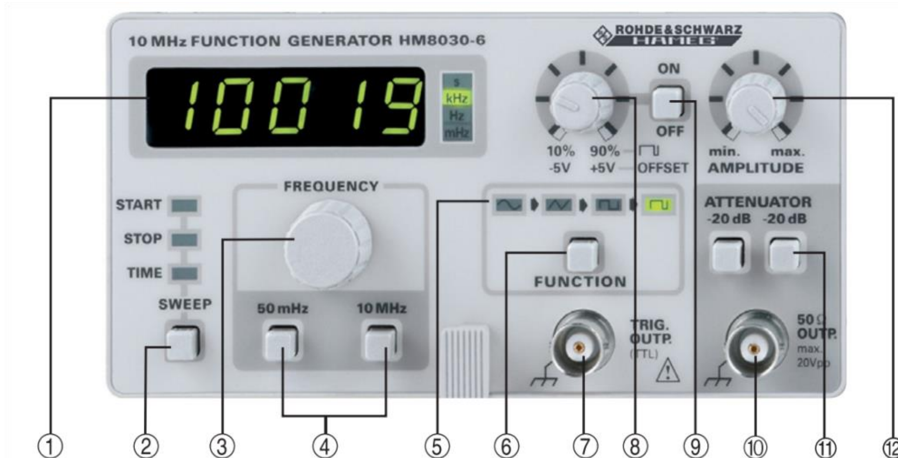


Figura 1.3 - Panoul frontal HM8030-6

1.4.1 COMENZI DE FUNCȚIONARE

- partea de **reglare a frecvenței** semnalului generat (**FREQUENCY**), cu 2 tipuri de reglaje: **în trepte** (butoanele **50 mHz și 10 MHz (4)** – ce micșorează, respectiv, măresc ordinul măsurătorii) și **un reglaj fin** (un buton rotativ **(3)**);

- partea de **reglare a amplitudinii** semnalului generat (**AMPLITUDE**), cu 2 tipuri de reglaje: **brut** (se modifică valoarea **atenuării** introdusă de aparat – 2 butoane de atenuare de **-20 dB** fiecare **(11)**) și un reglaj **fin (12)**; de asemenea, se poate regla **componenta continuă** a semnalului generat (**OFFSET (8)**) prin apăsarea butonului **ON/OFF (9)** și acționarea potențiometrului **(8)**;

- un buton pentru **selectarea formei** semnalului de ieșire (**FUNCTION (6)**) indicată de LED-urile **(5)**;

- **2 ieșiri de semnal**: una pentru **semnalul dorit de utilizator (10)**, având forma de undă și valorile reglate pentru **frecvență** și **amplitudine** (de ex. un semnal sinusoidal cu frecvența **20 Hz** și amplitudinea **4 V**) și o ieșire pentru semnal **de sincronizare TTL (7)**.

1.4.2 MODUL DE UTILIZARE

Pentru folosirea acestui generator, se **activează alimentarea grupului de module** prin **apăsarea** butonului **roșu** din mijlocul acestuia. Primul pas după deschidere constă în **selectarea funcției dorite** cu **(6)**, urmând **selectarea funcției butonului rotativ (8)** și apoi reglarea **amplitudinii** și **offsetului** sau **factorului de umplere**. Apoi, se va selecta **frecvența dorită** pentru **generare** folosind **(4)** pentru **domeniul de frecvență** și **(3)** pentru a ajunge cât mai aproape de **frecvența dorită**. În cazul în care semnalul dorit trebuie **atenuat**, se folosesc butoanele **(11)**. În cazul în care avem nevoie de un **semnal de sincronizare**, se folosește ieșirea **(7)** ce pune la dispoziție un **semnal TTL**. Ieșirea **(10)** va genera semnalul dorit continuu până la **tăierea alimentării** sistemului.

1.5 MULTIMETRUL DIGITAL PROGRAMABIL HAMEG HM8012

Definiție

Un multimetru este un instrument de măsurare electronic care combină mai multe funcții de măsurare într-o singură unitate. Acesta include caracteristici de bază, cum ar fi: capacitatea de a măsura tensiunea, curentul și rezistența. Multimetrele digitale (DMM, DVOM) afișează valoarea măsurată în cifre.

Caracteristici ale **HAMEG HM8012**:

- Display de 4 ¾ digiți cu 50000 unități;
- 42 domenii de măsurare;
- Domeniu automat sau manual;
- Între 3 și 6 măsurători pe secundă;
- Precizie de 0.05%;
- Rezoluții: **10 µV**, **10 nA**, **10 mΩ**, **0,01 dBm** și **0,1°**.

Figura 1.4 - Panoul frontal HM8012

1.5.1 COMENZI DE FUNCȚIONARE

- **(1) – Afișaj**
  Indică valoarea de măsurare cu o rezoluție de **4 ¾ digiți**, unde cea mai mare cifră folosită este **"5"**. Acesta va afișa, de asemenea, diverse **mesaje de avertizare**. Valoarea de măsurare va fi afișată cu puncte zecimale și polaritate.

- **(2) – LED martor continuitate**

- **(3) – Activare / Dezactivare martor sonor pentru continuitate**
  Activează/dezactivează semnalul acustic pentru funcția de continuitate

- **(4), (5), (7) și (9) – Conectori banană de 4mm, unde:**
  - 4 – conector pentru măsurarea a **maximum 10 A** în conjuncție cu intrarea **COM (7)**;
  - 5 – conector pentru măsurarea a **maximum 500 mA** în conjuncție cu intrarea **COM (7)**;
  - 7 – conector comun pentru **toate măsurătorile** ce are **potențial** apropriat cu masa (**0 V**) cantității măsurate;
  - 9 – conector pentru măsurarea **voltajelor**, **rezistențelor**, **temperaturilor** și **joncțiunilor de diode** în conjuncție cu intrarea **COM (7)**.

- **(6) – HOLD (LED)**
  LED martor ce indică faptul că valoarea indicată de afișaj **nu se mai actualizează** cu valoarea curentă. Această funcție este activată de butonul **(10)**.

- **(8) – OFFSET (LED)**
  LED martor ce indică faptul că afișajul **arată o măsurătoare relativă**. Valoarea indicată reprezintă **diferența** între valoarea de la intrare și cea prezentă la activarea modului **OFFSET**. Activarea acestei funcții se face prin apăsarea butonului **(10)**.
- **(11) și (12) – Domeniu**
  Butoane pentru schimbarea **domeniului de măsurare**.

- **(15) – Auto-Range**
  Buton pentru activarea / dezactivarea funcției de **auto-range**.

- **(16) – Unitatea de măsurare**
  Această zonă afișează **unitatea de măsură selectată** pentru funcția de măsurare.

- **(17) – AC/DC**
  Selectare pentru măsurare în domeniu **DC** sau **AC**.

- **(18) și (19) – Funcție măsurare**
  Selectarea tipului de măsurare.

1.5.2 MODUL DE UTILIZARE

Pentru a măsura un voltaj **se alimentează modulul** apăsând pe butonul **roșu** din mijlocul grupului de 2 module. Inițial, modulul **are toți martorii luminoși aprinși**, iar după terminarea inițializării aceștia **se sting**, rămânând **martorul „V" activat** în grupul **(16)**, iar valoarea voltajului măsurat este afișată pe **(1)**. După inițializare, **se conectează** conectorul **(7)** la **masa sistemului** unde se face măsurătoarea, iar **(9)** la punctul **unde se dorește a se măsura** voltajul. În cazul în care martorul **(14) nu este activat**, **se selectează** domeniul de măsură **manual** pentru obținerea preciziei dorite.

Pentru a măsura alte caracteristici ale semnalelor se folosesc butoanele **(18)** și **(19)** pentru a **schimba**, modificările fiind vizibile cu ajutorul martorilor din **(16)**. În cazul **măsurării** curentului dintr-un **circuit**, aparatul **se conectează în serie** pe latura unde se dorește a se măsura curentul, de această dată folosind conectorii **(7)** și **(4)** sau **(5)**, în funcție de precizia dorită.

Pentru a **păstra** valoarea măsurată se apasă **(10)** și se aprinde martorul **(6)**, iar pe **(1)** se oprește **actualizarea** cu noi date. În cazul măsurării unei valori **rezistive**, se poate activa funcția de **buzzer** cu butonul **(3)**, martorul **(2)** reflectând **starea funcției**. Astfel, când valoarea citită este **foarte aproape** de **0 Ω**, modulul **emite un sunet** până când valoarea **crește**. Un comportament asemănător poate fi observat și în timpul măsurării unei **diode** dacă funcția de **buzzer** este **activată**. În acest caz, se constată faptul că, dacă intrările **(7)** și **(9)** sunt în **scurt-circuit**, se aud **2 beep-uri scurte pe secundă** până când intrările nu mai sunt **scurtcircuitate** sau se oprește **buzzer-ul**.

1.6 FRECVENȚMETRUL NUMERIC 1.6 GHZ HAMEG HM8021-4

Cu ajutorul acestui modul se pot măsura **caracteristicile de timp** a semnalelor cu o frecvență de până la **1.6 GHz**.

Acesta prezintă caracteristicile:

- Domeniu de până la **1.6 GHz**;
- Senzitivitate **20 mV**;
- Funcții de măsurare;
- Trei perioade de eșantionare;
- Auto-trigger.

Figura 1.5 - Panoul frontal HM8021-4

1.6.1 COMENZI DE FUNCȚIONARE

- **(1) - OF (LED)**

LED martor ce este aprins când apare o depășire (overflow) a registrului intern de numărare a modulului. Aceasta depinde de perioada de eșantionare.

- **(2) - GT (GATE OPEN, LED)**

LED-ul martor GT este aprins atunci când intrarea este deschisă pentru măsurători.

- **(3) - Gate Time**

Butonul selectează perioada de eșantionare (0.1s, 1s, 10s), iar modificările pot fi observate la martorii de deasupra butonului.

- **(4) – Hold**

Apăsarea acestuia oprește actualizarea informației afișate până la apăsarea butonului *RESET*.

- **(5) – Function**

Martorii LED indică funcția de măsurare selectată cu ajutorul butoanelor.

- **(6) – Offset**

La apăsarea butonului, valoarea afișată devine noua valoare de referință.

- **(7) – Reset**

Buton ce resetează valoarea afișată cât și registrele interne ale aparatului.

- **(8) - Input C**

Intrare de semnal cu o frecvență de 100 MHz - 1,6 GHz, având o impedanță de 50Ω.

- **(9) – DC**

Buton ce permite **selectarea** semnalului de intrare ca fiind **AC** sau **DC**.

- **(10) - 1:20**

Selectează **atenuarea** semnalului de intrare. Semnalul de intrare este **atenuat** cu **26 dB**.

- **(11) – Auto**

Dacă acest buton este apăsat, **se ignoră ajustarea manuală** și **se caută automat** o valoare pentru **trigger** în funcție de tipul și caracteristicile semnalului.

- **(12) - Input A**

Pe această intrare se pot măsura semnale până la **150 MHz**, impedanța de intrare fiind **1 MΩ**.

- **(13) - Trigger Level**

Potențiometru de **ajustare** a trigger-ului. LED-ul martor **clipește** atunci când trigger-ul este **corect**.

- **(14) - Afișaj de 8 digiți plus exponent cu semn**

- **(15) - Hz (LED) / Sec (LED)**

Indică faptul că **se măsoară o frecvență**, respectiv indică faptul că **se măsoară un timp**.

1.7 GENERATORUL DE FORME DE UNDĂ SDG2000X

Generatorul de forme de undă **SDG2000X**, produs de **SIGLENT**, este un echipament **versatil** și **performant** destinat **generării de unde complexe** pentru o varietate de aplicații, inclusiv **cercetare**, **dezvoltare** și **testare**. Seria **SDG2000X** oferă o **precizie ridicată**, o **gamă variată** de forme de undă și **funcții avansate** pentru a răspunde cerințelor moderne din domeniul electronicii.

Acesta prezintă:

- **Canale duale:** permite generarea simultană de semnale diferite pe 2 canale;
- **Lățime de bandă extinsă:** Până la **120 MHz** pentru semnale sinusoidale;
- **Rezoluție ridicată: 16 biți** și o rată de eșantionare de până la **1,2 GSa/s**;
- **Tehnologii avansate:** Integrarea tehnologiilor **TrueArb** și **EasyPulse** pentru generarea precisă a undelor arbitrare și a impulsurilor;
- **Funcții multiple de modulare: AM**, **FM**, **PM**, **PWM**, **ASK**, **FSK**, **PSK** și altele;
- **Interfețe diverse**: **USB**, **LAN**, și opțional **GPIB**.

1.7.1 PANOUL FRONTAL

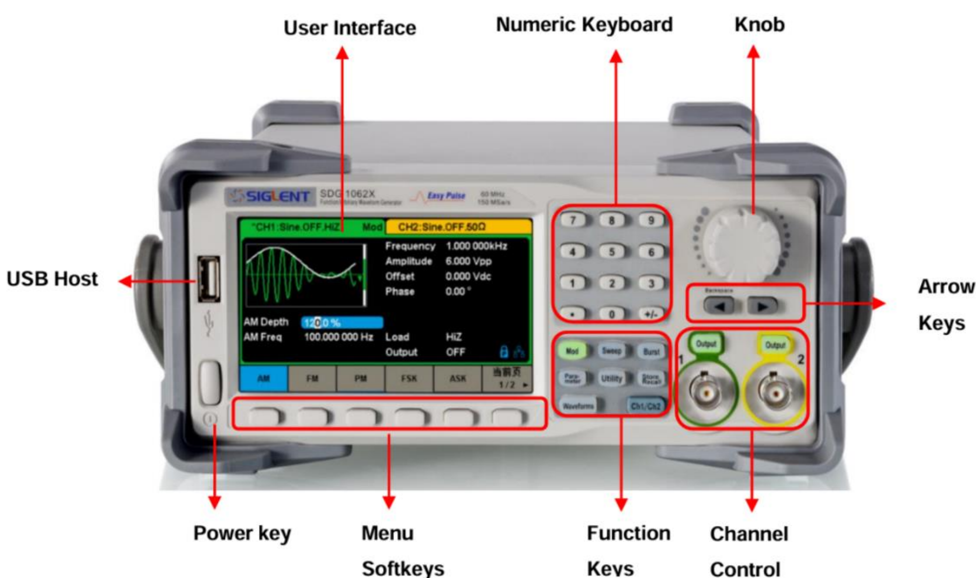


Figura 1.6 - Panoul frontal al generatorului SDG2000X

1.7.1.1 ECRAN TACTIL (USER INTERFACE)

Afișează parametrii curenți ai formei de undă selectate. Permite navigarea prin meniuri și configurarea parametrilor prin atingere. Secțiunile ecranului includ:

1.7.1.2 TASTE FUNCȚIONALE (MENU SOFTKEYS)

Oferă **acces** la diferite **funcții** și **meniuri** pentru **configurarea** semnalelor și setărilor.

1.7.1.3 TASTATURĂ NUMERICĂ (NUMERIC KEYBOARD)

Permite **introducerea precisă** a valorilor parametrilor pentru **frecvență**, **amplitudine**, **offset** și **fază**.

1.7.1.4 BUTON MULTIFUNCȚIONAL ROTATIV (KNOB)

Utilizează **rotirea** pentru **ajustarea valorilor parametrilor** și **apăsarea** pentru **confirmarea selecției**.

1.7.1.5 TASTE DIRECȚIONALE (ARROW KEYS)

Permit **navigarea** între **opțiuni** și **meniuri** pentru o configurare detaliată.

1.7.1.6 TASTE DE CONTROL AL CANALELOR (CHANNEL CONTROL)

Selectează și **configurează** parametrii canalelor **CH1** și **CH2**. De asemenea, **permite activarea** sau **dezactivarea ieșirii** fiecărui canal.

1.7.1.7 TASTE FUNCȚIONALE SUPLIMENTARE (FUNCTION KEYS)

Permit acces rapid la funcții precum **Utility**, **Mod** și **Waveforms** pentru configurarea avansată.

1.7.1.8 PORT USB HOST

Permite conectarea la dispozitive **USB** pentru **salvarea** sau **încărcarea datelor** de configurare.

1.7.1.9 BUTON DE ALIMENTARE (POWER KEY)

Controlează **alimentarea** echipamentului. LED-ul integrat indică **starea de funcționare**.

1.7.2 PANOUL POSTERIOR

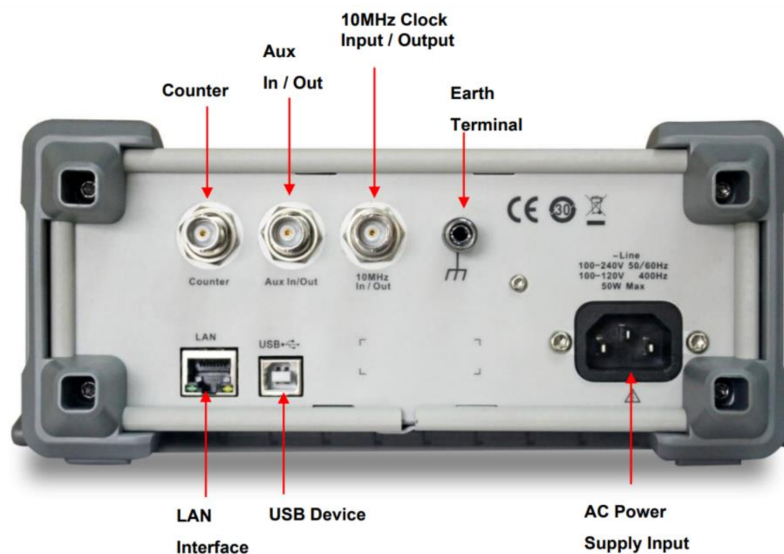


Figura 1.7 - Panoul posterior al generatorului SDG2000X

1.7.2.1 CONECTOR AUX IN/OUT

Permite utilizarea semnalelor de modulație externe sau sincronizarea cu alte dispozitive.

1.7.2.2 INTRARE/IEȘIRE CEAS 10 MHZ

Utilizat pentru sincronizarea cu alte echipamente prin semnal de ceas standard.

1.7.2.3 TERMINAL DE ÎMPĂMÂNTARE

Conectare la împământare pentru siguranță electrică.

1.7.2.4 CONECTOR LAN

Permite conectarea la o rețea locală pentru control și monitorizare de la distanță.

1.7.2.5 CONECTOR USB DEVICE

Asigură comunicarea cu un PC pentru configurare și control avansat.

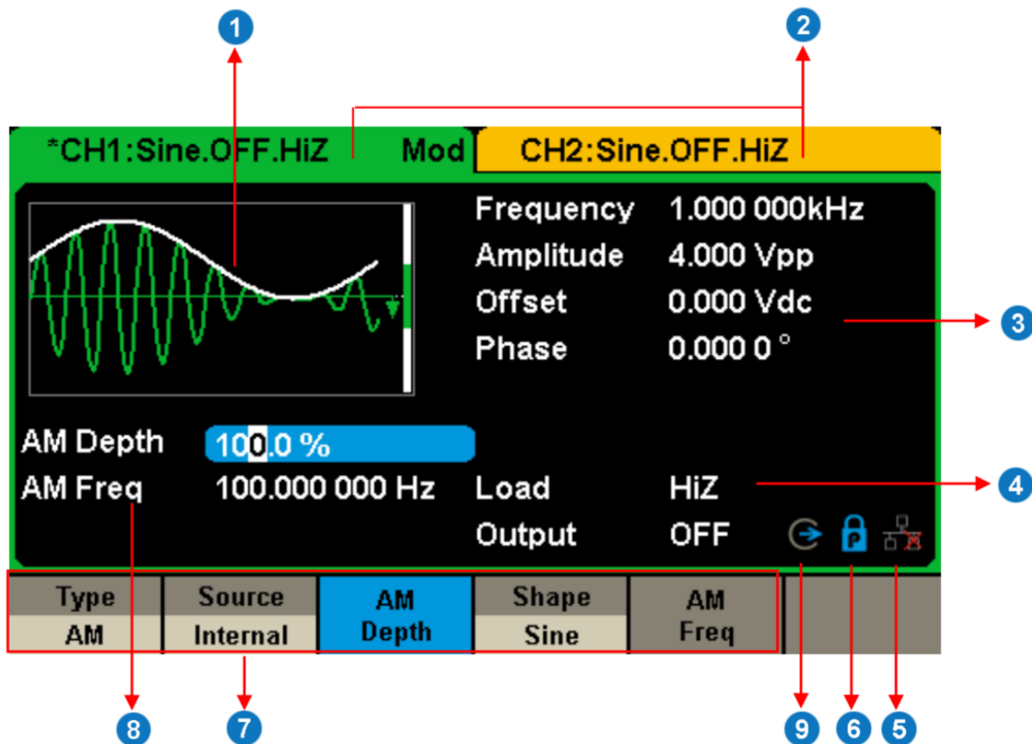
1.7.2.6 INTRARE ALIMENTARE AC

Conector pentru cablul de alimentare.

1.7.2.7 COUNTER (INTRARE SEMNAL)

Permite utilizarea funcției de frecvențmetru pentru măsurarea frecvenței semnalelor aplicate.

1.7.3 DISPLAY



Figură 1.8 - Display-ul generatorului SDG2000X

1.7.3.1 COMENZI DE FUNCŢIONARE:

- **(1) – Waveform Display Area**
  Afișează **forma de undă curentă** selectată pentru fiecare canal.

- **(2) – Channel Status Bar**
  Indică **starea selectată** și **configurația de ieșire** a canalelor.

- **(3) – Basic Waveform Parameters Area**
  Prezintă **parametrii curenți** ai formei de undă pentru fiecare canal. Aceștia pot fi **configurați** folosind **tastele numerice** sau **butonul rotativ**.

- **(4) – Channel Parameters Area**
  Afişează **setările de sarcină și ieșire** ale canalului selectat, precum:

  - **Load**
    Valoarea **sarcinii de ieșire** (**50 Ω** implicit, ajustabilă între **50 Ω** și **100 kΩ**).

  - **Output**
    Starea ieșirii canalului (**pornit** / **oprit**).

- **(5) – LAN Status Icon**
  Indică starea **conexiunii LAN**.

- **(6) – Mode Icon**
  Indică modul curent (**Independent** sau **Phase-locked**).

- **(7) – Menu**
  Afişează **meniul** corespunzător funcției selectate.

- **(8) – Modulation Parameters Area**
  Prezintă **parametrii** funcției de modulație curente.

- **(9) – Clock Source Icon**
  Prezintă starea sursei de clock: dacă aceasta este internă sau externă, or dacă sursa de clock nu este disponibilă ca sursă de clock externă.

1.7.4 MODUL DE UTILIZARE

Se pornește generatorul prin apăsarea **butonului de alimentare**, după care se selectează **canalul dorit** (**CH1** sau **CH2**). Apoi, se setează **forma de undă** prin apăsarea tastei **Waveforms** și selectarea tipului de semnal (**sinusoidal**, **dreptunghiular**, **triunghiular**, **zgomot**, **arbitrar**). Pentru ajustarea parametrilor, se folosește **butonul rotativ** sau **tastatura numerică** pentru configurarea **frecvenței**, **amplitudinii**, **offset-ului** sau **fazei**. Tot ce mai rămâne de făcut e activarea ieșirii prin apăsarea butonului **Output**, care activează semnalul.

1.8 SARCINA ELECTRONICĂ PROGRAMABILĂ SDL1000X

SDL1000X este o sarcină electronică programabilă, utilizată pentru **testarea surselor de alimentare**, a **bateriilor** și a altor **componente electronice**. Acesta oferă moduri de operare variate, incluzând **curent constant** (**CC**), **tensiune constantă** (**CV**), **rezistență constantă** (**CR**) și **putere constantă** (**CP**). Aparatul dispune de un afișaj **TFT-LCD** de **3.5 inch**, interfață ușor de utilizat și performanțe remarcabile pentru o gamă largă de aplicații.

Caracteristicile prezentate sunt:

- **Două versiuni: SDL1020X** (**150 V / 30 A**, **200 W**) și **SDL1030X** (**150 V / 30 A, 300 W**);
- **Rezoluție minimă de citire**: **0.1 mV / 0.1 mA**;
- **Moduri statice și dinamice**: CC, CV, CR, CP;
- Interfețe de comunicare **RS232 / USB / LAN** cu protocol **SCPI** integrat;
- **Funcții de protecție**: supracurent (**OCP**), supratensiune (**OVP**), supraîncălzire (**OTP**), polaritate inversă (**RPP**).

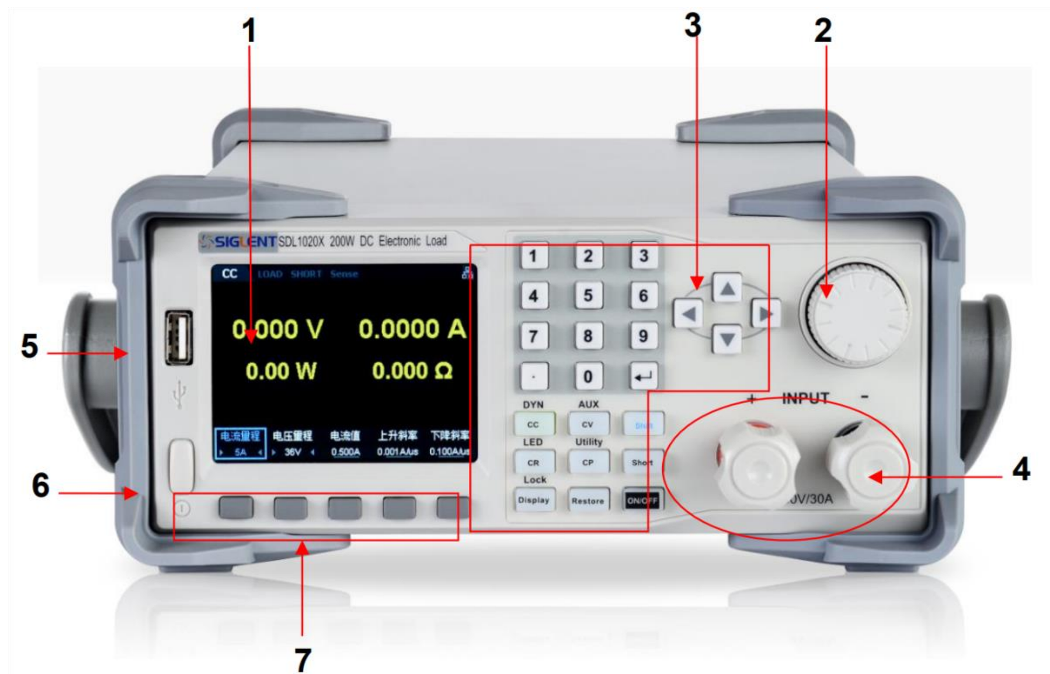


Figura 1.9 - Panoul frontal al sarcinii electronice SDL1000X

1.8.1 PANOUL FRONTAL

1.8.1.1 LCD

Afișează **sistemul de parametri**, **starea ieșirilor**, **undele grafice**, **opțiuni de meniu** și **mesaje informative**. Dimensiunea sa de **3.5 inch** permite vizualizarea clară a tuturor datelor esențiale.

1.8.1.2 KNOB ROTATIV

Buton utilizat pentru a ajusta **parametrii**. Rotirea **knob-ului** modifică **valoarea curentă** a parametrului selectat, iar **apăsarea** acestuia **confirmă selecția**.

1.8.1.3 BUTOANE FUNCȚIONALE PRINCIPALE

- **CC** (**Curent Constant**) – selectează modul de operare pentru menținerea unui **curent constant**.
- **CV** (**Tensiune Constantă**) – selectează modul pentru menținerea unei **tensiuni constante**.
- **CR** (**Rezistență Constantă**) – permite simularea unei **rezistențe constante**.
- **CP** (**Putere Constantă**) – activează modul de menținere a unei **puteri constante**.
- **Display** – afișează **formele de undă** și **informații** despre starea dispozitivului.
- **Shift** – oferă acces la **funcțiile alternative** ale celorlalte butoane (ex. **modul dinamic**, **LED**).

1.8.1.4 TERMINALE DE INTRARE

Intrările de semnal sunt poziționate pe partea **frontală** și permit **conectarea** circuitului de testare.

1.8.1.5 INTERFAȚĂ USB

Port USB pentru conectarea **dispozitivelor de stocare** sau pentru **transferul de date**.

1.8.1.6 BUTON DE ALIMENTARE

Situat în partea **inferioară** a panoului **frontal**, permite **pornirea** și **oprirea** dispozitivului.

1.8.1.7 FUNCTION KEY

Tastele situate sub **ecranul LCD** sunt utilizate pentru a alege diferite funcții.

1.8.2 PANOUL POSTERIOR

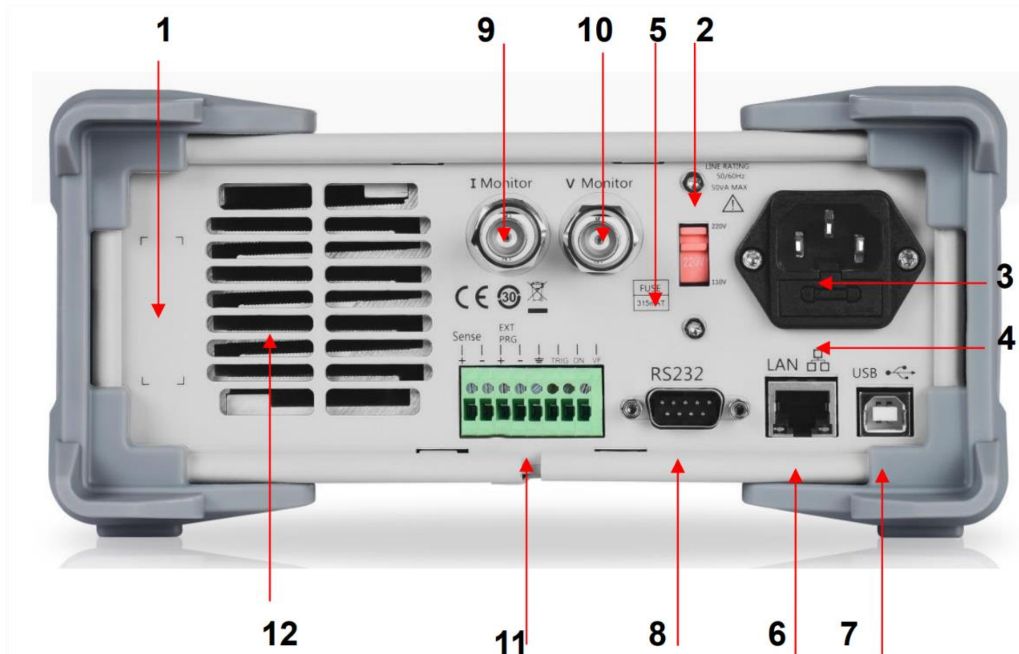


Figura 1.8 - Panoul posterior al sarcinii electronice SDL1000X

1.8.2.1 WARNING MESSAGE

Avertismente despre împământarea instrumentului și alte informații importante.

1.8.2.2 DESCRIEREA TENSIUNII DE INTRARE

Frecvența și tensiunea sursei de alimentare AC trebuie să se potrivească cu specificația siguranței.

1.8.2.3 CONECTOR DE ALIMENTARE AC

Permite conectarea cablului de alimentare pentru sursa de curent alternativ.

1.8.2.4 FUSE (SIGURANȚĂ)

Siguranța specificată trebuie să fie ajustată pentru tensiunea de intrare. Trebuie să consultați descrierea tensiunii AC de intrare.

1.8.2.5 SELECTOR TENSIUNE DE ALIMENTARE

Asigură compatibilitatea cu diferite tensiuni de intrare (110 V/220 V).

1.8.2.6 PORT LAN

Pentru conectarea la rețea, oferind acces la funcții de control remote prin protocol SCPI.

1.8.2.7 USB DEVICE

Conectează instrumentul (ca dispozitiv controlat) la un computer/controller extern prin USB.

1.8.2.8 PORT RS232

Utilizat pentru comunicare serială cu alte dispozitive.

1.8.2.9 TERMINAL DE DETECTARE A CURENTULUI

Pentru a observa intrarea curentului în SDL conectând terminalul de detectare a curentului la un instrument de măsurare (osciloscop, multimetru digital) pentru a analiza modificarea curentului de intrare în funcție de timp.

1.8.2.10 TERMINAL DE DETECTARE A TENSIUNII

Pentru a observa tensiunea de intrare în SDL conectând terminalul de detectare a tensiunii la un instrument de măsurare (osciloscop, DMM) pentru a analiza modificarea tensiunii de intrare în funcție de timp.

1.8.2.11 TERMINAL DE DETECTARE / CONTROL EXTERN / IEȘIRE PWM

Se selectează portul corespunzător pentru funcția dorită.

1.8.2.12 VENTILATOR DE RĂCIRE (FAN)

Menține temperatura optimă de funcționare a dispozitivului, prevenind supraîncălzirea.

1.8.3 MODURI DE OPERARE

- **Curent Constant (CC)**: sarcina absoarbe un **curent fix**, indiferent de tensiunea de intrare.
- **Tensiune Constantă (CV)**: sarcina reglează curentul pentru a menține o **tensiune fixă** la intrare.
- **Rezistență Constantă (CR)**: simulează un **rezistor constant**, schimbând curentul proporțional.
- **Putere Constantă (CP)**: sarcina ajustează curentul pentru a menține o **putere constantă**.

1.8.4 MODUL DE UTILIZARE

În primul rând, se **verifică** integritatea fizică a **dispozitivului** și **accesoriile incluse**. Apoi, se **verifică** că selectorul de tensiune de pe panoul din spate **corespunde** tensiunii de alimentare. În continuare, se **conectează** aparatul la sursa de alimentare și **se pornește** utilizând butonul de alimentare. Se **selectează** modul dorit (**CC**, **CV**, **CR**, **CP**) folosind butoanele de pe panoul frontal, după care se **ajustează** parametrii corespunzători (**curent**, **tensiune**, **rezistență** sau **putere**) utilizând knob-ul rotativ. În final, se **activează** protecțiile pentru **supracurent**, **supratensiune** sau **polaritate inversă**, conform cerințelor aplicației și se **monitorizează** parametrii afișați pentru a **preveni daunele**.

1.9 MULTIMETRUL DIGITAL SDM3065X

Multimetrul digital **SDM3065X** este un instrument de precizie înaltă, proiectat pentru măsurători variate, inclusiv **tensiuni**, **curenți**, **rezistențe**, **capacități** și **temperaturi**. Cu un ecran color **TFT-LCD** de **4.3 inch** și rezoluție de **6 ½ cifre**, acest dispozitiv suportă mai multe funcții matematice și moduri de afișare. Este ideal pentru aplicații care necesită măsurători de precizie și automate.

Printre cele mai bune caracteristici se numără:

- Afișaj color **TFT-LCD** de **4.3 inch**, cu rezoluție de **480x272**;
- Rezoluție maximă de **6 ½ cifre**;
- Viteză de măsurare de până la **150 citiri / secundă**;
- **Funcții de măsurare**: tensiune **DC/AC**, curent **DC/AC**, rezistență (**2 și 4 fire**), capacitate, frecvență și perioadă, continuitate, diodă și temperatură;
- Memorie internă de **1 Gb** pentru stocarea fișierelor;
- **Interfețe de comunicație multiple**: **USB Device & Host**, **LAN** și **USB-GPIB** (opțional);
- Sistem de ajutor integrat și suport pentru comenzi **SCPI**.

1.9.1 PANOUL FRONTAL

Figura 1.9 - Panoul frontal al multimetrului digital SDM3065X

1.9.1.1 (A) – INTERFAȚĂ USB

Utilizează un **port USB** pentru **stocarea datelor** sau pentru **actualizări de firmware**.

1.9.1.2 (B) – BUTON DE ALIMENTARE

Activarea sau **dezactivarea** dispozitivului.

1.9.1.3 (C) – ECRAN LCD

Afișează meniurile **funcțiilor**, setările **parametrilor de măsurare**, starea **sistemului** și **mesaje de avertizare**. Dimensiunea de **4.3 inch** și rezoluția ridicată asigură o vizibilitate clară.

1.9.1.4 (D) – BUTOANE DE OPERARE A MENIULUI

Permite accesarea **meniurilor** corespunzătoare **funcțiilor** afișate pe ecran.

1.9.1.5 (E) – TASTE FUNCȚIONALE

- **DCV / ACI** – măsurarea tensiunii și curentului alternativ;
- **DCI** – măsurarea curentului continuu;
- **Rezistență 2 fire / 4 fire** – pentru măsurarea precisă a rezistenței;
- **Capacitate / Frecvență** – testarea capacității sau a frecvenței;
- **Continuitate / Diode** – testarea conexiunilor și a diodelor;
- **Temperatură** – măsurarea temperaturii cu senzori termocuplu sau termistor.

1.9.1.6 (F) – BUTOANE DE NAVIGARE

- Permit **selectarea** și **ajustarea** parametrilor de măsurare;
- Opțiuni de navigare în **sus / jos**, **stânga / dreapta**.

1.9.1.7 (G) – CONECTORI DE INTRARE

Terminalele **HI** și **LO** sunt utilizate pentru conexiuni standard de măsurare, iar terminalele **HIsense** și **LOsense** sunt dedicate măsurătorilor de rezistență în **4 fire**.

1.9.2 PANOUL POSTERIOR

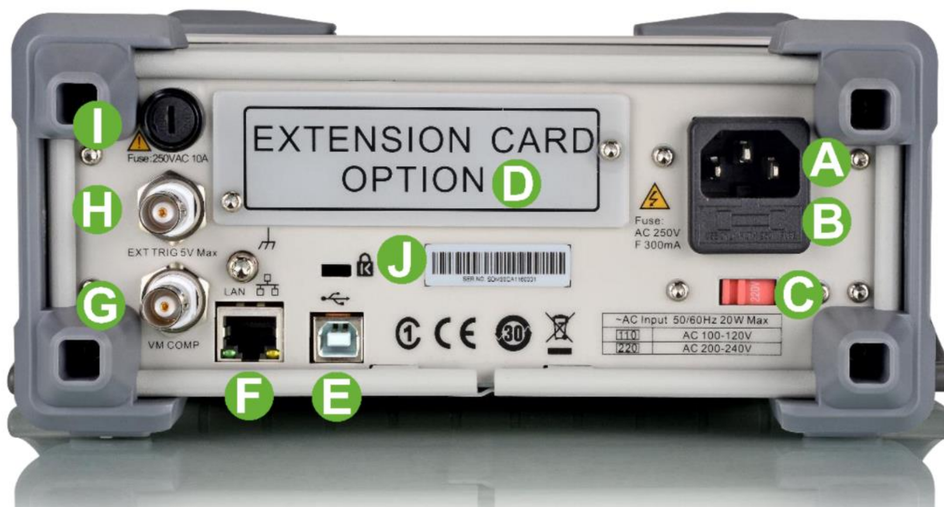


Figura 1.10 - Panoul posterior al multimetrului digital SDM3065X

1.9.2.1 (A) – CONECTOR DE ALIMENTARE AC

Permite conectarea **cablului de alimentare** pentru sursa de curent alternativ.

1.9.2.2 (B) – SIGURANȚĂ DE ALIMENTARE

1.9.2.3 (C) – SELECTOR TENSIUNE DE ALIMENTARE

Configurabil pentru **110 V** sau **220 V**, în funcție de cerințele rețelei electrice locale. Selectorul este poziționat lângă conectorul de alimentare și trebuie ajustat manual.

1.9.2.4 (D) – CARD DE INSPECȚIE (INSPECTION CARD): OPȚIONAL

În instrument poate fi instalat un modul opțional de achiziție de date cu 16 canale.

1.9.2.5 (E) – PORT USB (DEVICE ȘI HOST)

Facilitează **transferul de date** și **actualizările de firmware** sau **utilizarea în aplicații PC**.

1.9.2.6 (F) – PORT LAN

Utilizat pentru conectarea la rețea pentru **control de la distanță** prin protocoale standard.

1.9.2.7 (G) – VMC OUTPUT (COMPARATOR DE TENSIUNE)

Terminal utilizat pentru aplicații speciale ce necesită **măsurători de înaltă precizie**.

1.9.2.8 (H) – EXT TRIG (DECLANȘARE EXTERNĂ)

Permite **sincronizarea cu alte echipamente** prin semnal de declanșare extern de până la **5 V**.

1.9.2.9 (I) – SIGURANȚA PENTRU CURENTUL DE INTRARE

Asigură protecție pentru **alimentarea circuitelor interne** (**250 V / 300 mA**). Se verifică periodic starea lor și se înlocuiesc dacă este necesar.

1.9.2.10 (J) – ORIFICIU DE BLOCARE A INSTRUMENTULUI (INSTRUMENT LOCKHOLE)

Se poate utiliza blocarea de siguranță pentru a bloca multimetrul într-un loc fix, dacă este necesar.

1.9.3 MODUL DE UTILIZARE

Se **verifică** integritatea fizică a dispozitivului și accesoriile incluse. Apoi, se **selectează** tensiunea corespunzătoare (**110 V / 220 V**) folosind selectorul din panoul din spate, după care se **conectează** cablul de alimentare la sursa electrică și se **pornește** dispozitivul utilizând butonul de alimentare. Se **selectează** funcția dorită (**DCV**, **ACI**, **DCI**, etc.) utilizând tastele funcționale, se **conectează** terminalele corespunzătoare și se **ajustează** intervalul de măsurare (**Range**) prin butoanele dedicate. În cele din urmă, se **asigură** respectarea limitărilor de protecție ale terminalelor de intrare și se **verifică** întreruperile sau suprasarcinile pentru a preveni **deteriorarea dispozitivului**.

1.10 OSCILOSCOPUL DIGITAL PROGRAMABIL SDS2000X PLUS

Osciloscopul digital programabil **SDS2000X PLUS** este un echipament avansat, utilizat pentru **captarea**, **vizualizarea** și **analizarea** semnalelor electrice. Acesta dispune de un ecran tactil de **10.1 inch**, cu rezoluție mare, și suportă funcții precum **măsurători automate**, **decodare** de semnale seriale și **analiză matematică** avansată. Este ideal pentru aplicații în cercetare și dezvoltare, întreținere industrială și învățământ tehnic.

Caracteristici principale:

- Lățime de bandă de până la **500 MHz**;
- **Rata de eșantionare**: **2 GSa/s**;
- Memorie de captură de până la **200 Mpts**;
- Afișaj tactil de **10.1 inch** cu rezoluție **1024x600**;
- Moduri de declanșare avansate: **Edge**, **Pulse**, **Slope**, **Video**, etc.;
- **Funcții de analiză**: **FFT**, decodare **I^{2}C**, **SPI**, **UART/RS232**, **CAN**, **LIN**;
- Interfețe multiple: **USB**, **LAN**, **HDMI**, **AUX**;
- Mod de redare a istoriei semnalului și funcții de salvare rapidă.

1.10.1 PANOUL FRONTAL

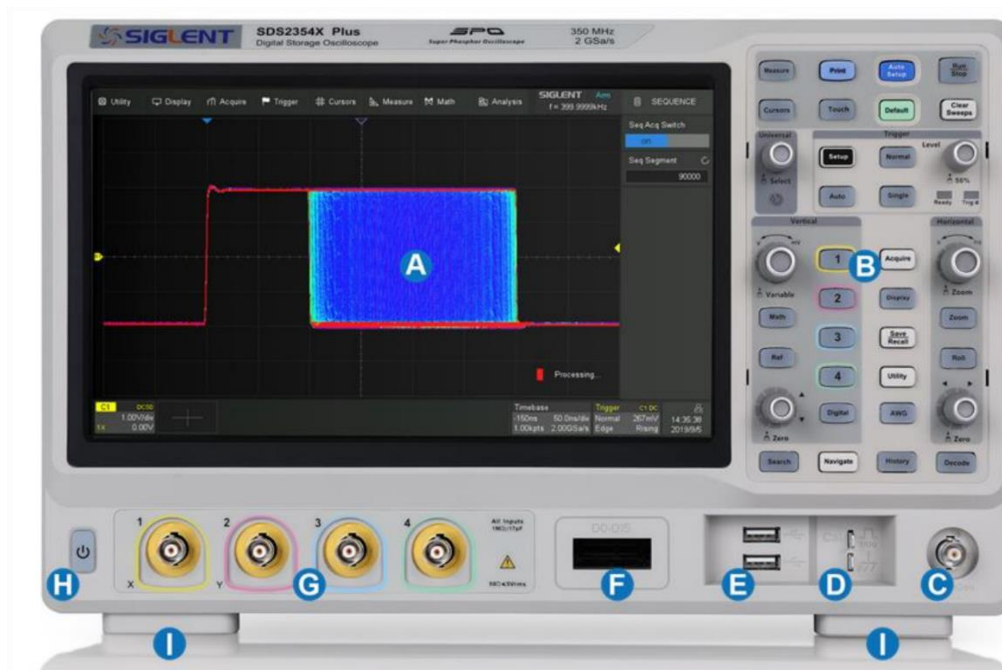


Figura 1.11 - Panoul frontal al osciloscopului digital SDS2000X PLUS

1.10.1.1 (A) – ECRAN TACTIL

Zona principală de afişare care permite utilizatorului să **vizualizeze** şi să **manipuleze** semnalele capturate. Permite operarea prin **atingere**, facilitând **navigarea** şi **ajustările parametrilor**.

1.10.1.2 (B) – PANOUL FRONTAL (FRONT PANEL)

Figura 1.12 - Panoul frontal cu butoane pentru SDS2000X Plus

Panoul frontal este conceput pentru a opera funcţiile de bază fără a fi nevoie să deschideţi meniul software. Majoritatea comenzilor de pe panoul frontal duplică funcţionalitatea disponibilă prin intermediul ecranului tactil, dar operarea se realizează mai rapid. Toate butoanele de pe panoul frontal sunt multifuncţionale. Pot fi apăsate, precum şi rotite. Apăsarea unui buton recheamă rapid o funcţie specifică, ceea ce este indicat de serigrafia de lângă buton.

1.10.1.3 (C) – WAVEGEN

Aceasta este ieşirea încorporată pentru generatorul de forme de undă.

1.10.1.4 (D) – COMPENSARE SONDĂ / TERMINAL DE MASĂ

Furnizează o undă pătrată de 0-3V, 1kHz pentru compensarea sondelor.

1.10.1.5 (E) – PORTURI GAZDĂ USB (USB HOST PORTS)

Se pot conecta dispozitive de stocare USB pentru transfer de date sau un mouse/tastatură USB pentru control.

1.10.1.6 (F) – CONECTOR INTRARE DIGITALĂ (DIGITAL INPUT CONNECTOR)

Primeşte semnale digitale de la sonda digitală SPL2016.

1.10.1.7 (G) – CONECTORI INTRARE ANALOGICĂ (ANALOG INPUT CONNECTORS)

1.10.1.8 (H) – ÎNTRERUPĂTOR DE ALIMENTARE (POWER SWITCH)

1.10.1.9 (I) – PICIOARE DE SPRIJIN (SUPPORTING LEGS)

Se reglează picioarele de sprijin pentru a le utiliza ca suporturi, înclinând osciloscopul pentru o poziționare stabilă.

1.10.2 PANOUL POSTERIOR

Figura 1.13 - Panoul posterior al osciloscopului digital SDS2000X PLUS

1.10.2.1 (A) – AUXILIARY OUT

Generează indicatorul de declanșare (trigger). Când funcția Pass/Fail este activată, generează semnalul de pass/fail.

1.10.2.2 (B) – EXT TRIGGER INPUT

Intrare pentru semnal de declanșare externă (până la **5 V**).

1.10.2.3 (C) – PORT LAN

Permite conectarea dispozitivului la rețea pentru control de la distanță.

1.10.2.4 (D) – PORT USB

Conector **USB** pentru conectarea la PC pentru **transfer de date** sau **control**.

1.10.2.5 (E) – CONECTOR DE ALIMENTARE AC

Intrare pentru cablul de alimentare, compatibil cu tensiuni de **100-240 V**.

1.10.2.6 (F) – MÂNER

1.10.3 PANOU FRONTAL CU BUTOANE

1.10.3.1 CONTROL VERTICAL

Figura 1.14 - Butoanele de control vertical pentru SDS2000X Plus

- **(A)** – Când un canal este dezactivat, apăsați butonul canalului pentru a-l activa. Când canalul este activat, dar inactiv, apăsați butonul pentru a-l activa. Când canalul este activat și funcțional, apăsați butonul pentru a-l dezactiva.
- **(B)** – Apăsați butonul pentru a activa/dezactiva canalul digital și a deschide caseta de dialog DIGITAL. Apăsați din nou pentru a dezactiva canalele digitale.
- **(C)** – Apăsați butonul pentru a activa/dezactiva funcția matematică și a deschide caseta de dialog MATH. Apăsați din nou pentru a dezactiva funcția matematică.
- **(D)** – Apăsați butonul pentru a activa/dezactiva funcția de referință și a deschide caseta de dialog REF. Apăsați din nou pentru a dezactiva funcția de referință.
- **(E)** – Canalele analogice (C1-C4), canalele digitale (D), funcțiile matematice (F1-F2) și referințele (Ref) împart același buton rotativ vertical. Rotiți butonul pentru a ajusta scara verticală (volți/diviziune). Apăsați pentru a alterna între ajustări grosiere și fine. Când canalul digital este activ, rotiți butonul pentru a schimba canalul digital selectat.
- **(F)** – Canalele analogice (C1-C4), canalele digitale (D), funcțiile matematice (F1-F2) și referințele (Ref) împart același buton rotativ de decalaj. Rotiți butonul pentru a ajusta decalajul DC sau poziția verticală a canalului. Apăsați pentru a seta decalajul la zero. Când canalul digital este activ, rotiți butonul pentru a schimba poziția canalului digital selectat.

1.10.3.2 CONTROL ORIZONTAL

Figura 1.15 - Butoanele de control orizontal pentru SDS2000X Plus

- **(A)** – Rotiți pentru a ajusta scara orizontală (timp/diviziune). Apăsați pentru a activa modul Zoom. Apăsați din nou pentru a ieși din modul Zoom.
- **(B)** – Apăsați pentru a activa modul Zoom. Apăsați din nou pentru a ieși din modul Zoom.
- **(C)** – Apăsați pentru a activa derularea orizontală (Horizontal Roll). Apăsați din nou pentru a ieși din modul Roll. La setări de bază de timp mai mari de 50 ms/diviziune , se recomandă setarea osciloscopului în modul Roll, astfel încât forma de undă să fie afișată în timp real.
- **(D)** – Rotiți pentru a ajusta întârzierea declanșării (trigger delay). Apăsați pentru a seta întârzierea declanșării la zero.

1.10.3.3 CONTROL DECLANȘARE (TRIGGER CONTROL)

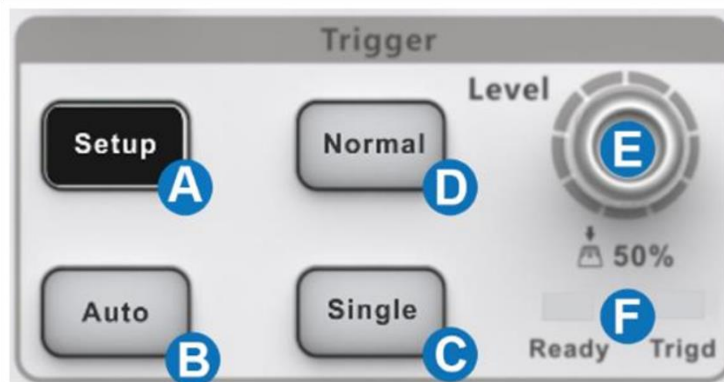


Figura 1.16 - Butoanele de control trigger pentru SDS2000X Plus

- **(A)** – **Setup**: deschide caseta de dialog pentru setările declanșării.
- **(B)** – **Auto**: declanșări după o perioadă prestabilită dacă nu apare o declanșare validă.
- **(C)** – **Single**: declanșări o singură dată când toate condițiile sunt îndeplinite.
- **(D)** – **Normal**: declanșări repetate când toate condițiile sunt îndeplinite.
- **(E)** – **Ajustare nivel trigger**: Apăsați pentru a seta nivelul la 50% din forma de undă.
- **(F)** – **Indicator stare declanșare**: Gata (Ready) sau Declanșat (Trig'd).

BUTON PORNIRE/OPRIRE (RUN/STOP BUTTON)

Apăsați butonul pentru a comuta starea de achiziție între Pornire (Run) și Oprire (Stop). Când starea este Pornire, butonul este iluminat în galben. Când starea este Oprire, butonul este iluminat în roșu.

1.10.3.4 BUTON AUTO SETUP

Osciloscopul va seta automat scala verticală, scala orizontală și nivelul de declanșare în funcție de semnalul de intrare pentru a obține afișarea optimă a formei de undă. De asemenea, puteți efectua o operațiune de configurare automată urmând pașii **Trigger > Auto Setup**.

1.10.4 DISPLAY

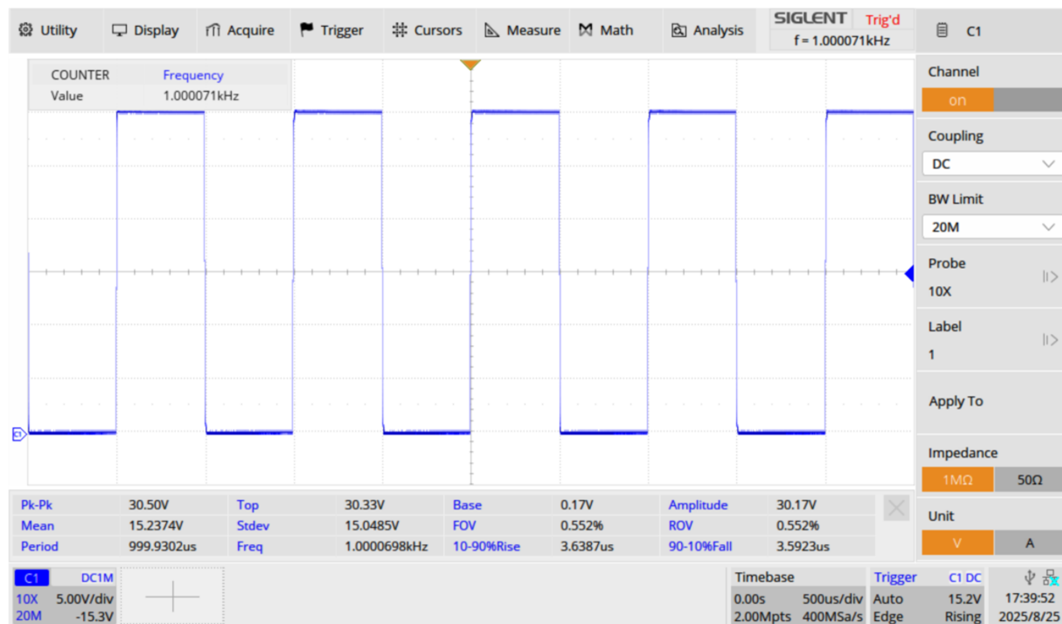


Figura 1.17 – Captură de osciloscop: Vizualizarea semnalului

Imaginea de mai sus prezintă ecranul principal al osciloscopului, unde se pot distinge clar **Bara de Menu**, incluzând opțiuni precum **Utilitar (Utility)**, **Afișaj (Display)**, **Achiziție (Acquire)** și altele. Prin accesarea acestor opțiuni, utilizatorul poate configura funcționalitățile osciloscopului și ajusta modul de vizualizare și analiză a semnalelor.

Zona centrală a ecranului este dominată de grila de afișare, esențială pentru interpretarea vizuală a formelor de undă. Aici, semnalul achiziționat de Canalul 1 este vizibil, iar grila permite o estimare rapidă a amplitudinii (în volți/diviziune) și a perioadei (în timp/diviziune).

Panoul lateral din dreapta conține setările specifice canalului selectat (în acest caz, C1). Acestea includ:

- **Canal (Channel)**: Indicarea canalului activ.
- **Cuplaj (Coupling)**: Selecția între DC (curent continuu), AC (curent alternativ) sau GND (masă) pentru semnalul de intrare.
- **Limitare Bandă (BW Limit)**: Opțiuni pentru filtrarea benzii de frecvență (ex: 20M pentru 20 MHz).
- **Sondă (Probe)**: Configurarea raportului de atenuare a sondei (ex: 10X).
- **Impedanță (Impedance)**: Setări pentru impedanța de intrare (1MΩ sau 50Ω).

Zona inferioară a ecranului este dedicată măsurătorilor automate și indicatorilor de stare.
Bara de stare de jos completează informațiile cu detalii despre ***Baza de Timp (Timebase),*** modul și nivelul de declanșare (Trigger), precum și data și ora curente. Indicatorul "Trig'd" din partea superioară dreaptă confirmă o declanșare reușită.

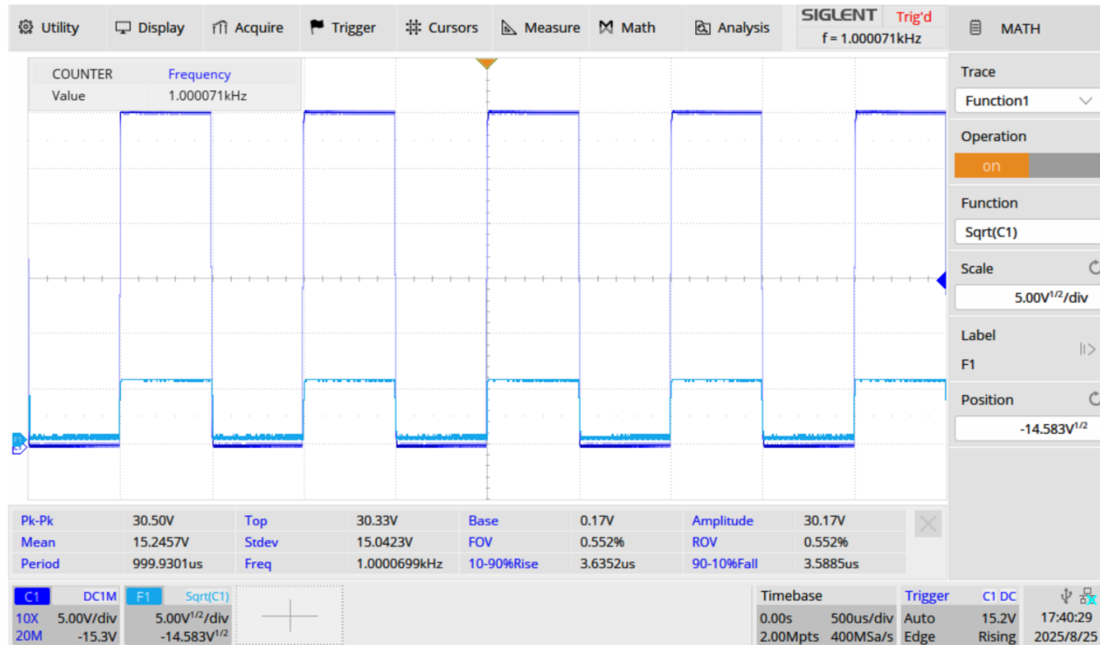


Figura 1.18 – Captură de osciloscop: Funcția Math

Figura prezintă interfața grafică a osciloscopului SDS2000X Plus în modul de operare cu funcții matematice. Această captură ilustrează aplicarea unei operații matematice pe un semnal achiziționat și afișarea rezultatului.

Zona centrală a ecranului afișează grila de achiziție, pe care sunt vizibile două forme de undă:

- **Semnalul original (Galben)**: acesta este semnalul de intrare, provenind de pe Canalul 1 (C1), după cum sugerează denumirea funcției matematice.
- **Semnalul rezultat al funcției matematice (Portocaliu)**: aceasta este forma de undă generată în urma aplicării operației matematice.

Panoul lateral din dreapta este configurat pentru setările funcției matematice. Această secțiune permite utilizatorului să definească și să controleze operația matematică:

- **Operație (Operation)**: indicat ca "on", semnalizând că funcția este activă.
- **Funcție (Function)**: specifică operația matematică aplicată. În acest caz, este setată la "Sqrt(C1)", indicând extragerea rădăcinii pătrate a semnalului de pe Canalul 1.
- **Scală (Scale)**: ajustează scara verticală a formei de undă rezultate, setată la "5.00V^2/div". Rețineți că unitatea este V^2 (volți pătrați) datorită operației de rădăcină pătrată.
- **Etichetă (Label)**: denumirea atribuită funcției matematice, în acest caz "F1".
- **Poziție (Position)**: setează decalajul vertical al formei de undă F1, indicat la "-14.583V^2".

1.10.5 MODUL DE UTILIZARE

În primul rând, se **verifică** dacă dispozitivul și accesoriile sunt **intacte**. Apoi, se **conectează** cablul de alimentare la priza corespunzătoare și se **pornește** osciloscopul folosind butonul de alimentare. Se **selectează** canalul dorit și se **ajustează** scala verticală și poziția, după care se **conectează** sonda la semnalul de test și se **calibrează** dacă este necesar. În continuare, se **ajustează** nivelul de declanșare și se **selectează** sursa de semnal pentru a asigura capturarea stabilă. Se **utilizează** funcțiile automate pentru a analiza caracteristicile semnalului, iar în final se **salvează** rezultatele pe un dispozitiv **USB** sau prin **LAN**.

1.11 SURSA DE ALIMENTARE PROGRAMABILĂ SPD3303X

SPD3303X / 3303X-E este o sursă de alimentare programabilă, versatilă și multifuncțională, concepută pentru aplicații de **testare** și **dezvoltare** în domeniul electronicii. Dispozitivul oferă **3 ieșiri independente**, **2 seturi de tensiune** ajustabilă și o **ieșire fixă** selectabilă de **2.5 V**, **3.3 V** sau **5 V**. De asemenea, include protecții pentru **scurtcircuit** și **supraîncărcare**, fiind un instrument de încredere pentru diverse aplicații profesionale.

Caracteristici principale:

- Afișaj color **TFT** de **4.3 inch**;
- 3 ieșiri independente, cu putere totală de până la **195 W**;
- **Moduri de operare**: **independent**, **paralel** și **serie**;
- Protecții pentru **supratensiune**, **supracurent** și **scurtcircuit**;
- **Interfețe de comunicație**: **USB**, **LAN**;
- **Funcții avansate**: afișare **undă**, **timer**, **salvare** și **apelare setări**.

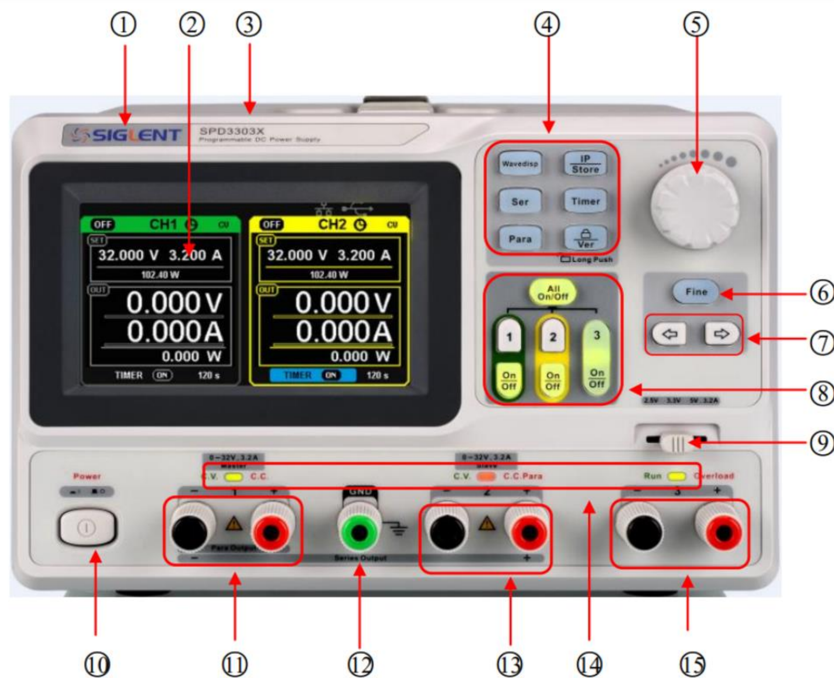


Figura 1.19 – Panoul frontal al sursei SPD3303X

1.11.1 PANOUL FRONTAL

1.11.1.1 (1) - LOGO

1.11.1.2 (2) - DISPLAY AREA

Prezintă parametrii de **tensiune**, **curent** și **putere**, precum și **graficele undelor semnalelor**.

1.11.1.3 (3) - MODELUL INSTRUMENTULUI

1.11.1.4 (4) - BUTON DE CONFIGURARE A PARAMETRILOR SISTEMULUI

Permite accesul la **meniul de configurare** a dispozitivului.

1.11.1.5 (5) - KNOB MULTIFUNCȚIONAL

Utilizat pentru **ajustarea valorilor** parametrilor selectați și **navigarea în meniu**.

1.11.1.6 (6) - BUTON DE AJUSTARE FINĂ (FINE ADJUST)

Permite **modificări precise** ale valorilor parametrilor.

1.11.1.7 (7) - BUTOANE DIRECȚIONALE

Pentru deplasarea **cursorului** în meniuri.

1.11.1.8 (8) - BUTON DE CONTROL AL CANALELOR

- **CH1** – activează sau selectează **canalul 1**;
- **CH2** – activează sau selectează **canalul 2**;
- **CH3** – activează sau selectează **canalul 3**;
- **ON / OFF** – activează sau dezactivează ieșirea **canalului selectat**.

1.11.1.9 (9) - SWITCH DIP CH3

1.11.1.10 (10) - COMUTATOR DE ALIMENTARE (POWER SWITCH)

1.11.1.11 (11) - TERMINALUL DE IEȘIRE PENTRU CANALUL 1 (CH1 OUTPUT TERMINAL)

1.11.1.12 (12) - GROUND TERMINAL

1.11.1.13 (13) - TERMINALUL DE IEȘIRE PENTRU CANALUL 2 (CH2 OUTPUT TERMINAL)

1.11.1.14 (14) - INDICATORI CV/CC

Afișează starea de operare a fiecărui canal: **tensiune constantă** (**CV**) sau **curent constant** (**CC**).

1.11.1.15 (15) - TERMINALUL DE IEȘIRE PENTRU CANALUL 3 (CH3 OUTPUT TERMINAL)

1.11.2 PANOUL POSTERIOR

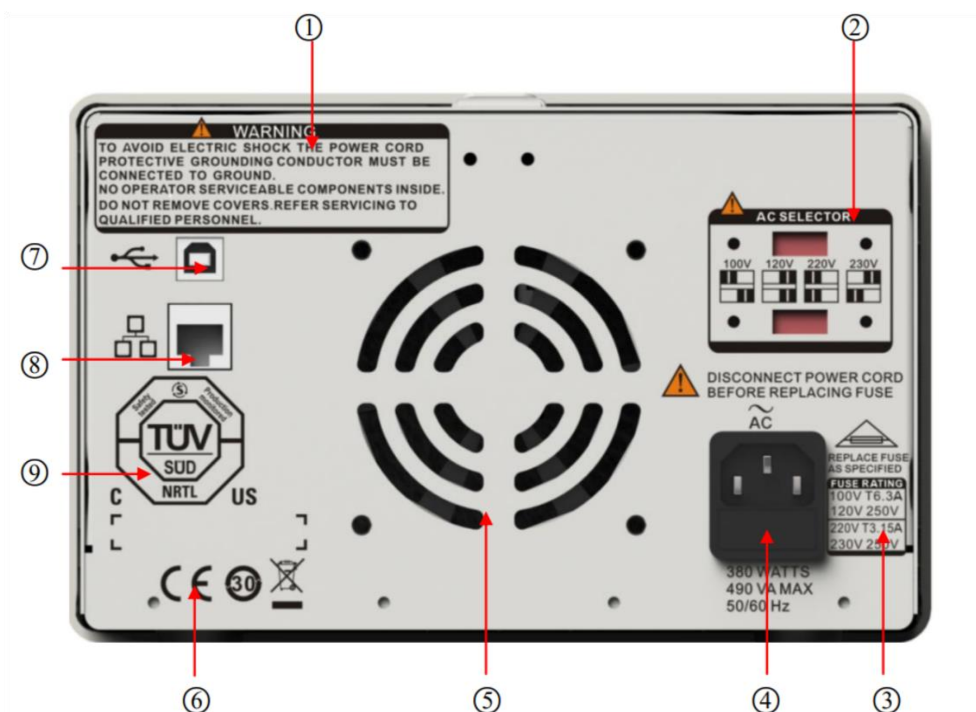


Figura 1.20 – Panoul posterior al sursei SPD3303X

1.11.2.1 (1) – MESAJ DE AVERTIZARE (WARNING MESSAGE)

1.11.2.2 (2) – COMUTATPORUL DIP PENTRU ALIMENTAREA AC ȘI IDENTIFICAREA ACESTUIA

1.11.2.3 (3) – DESCRIEREA TENSIUNII DE INTRARE AC

1.11.2.4 (4) – PRIZA DE ALIMENTARE AC

1.11.2.5 (5) – ORIFICIILE DE AERISIRE ALE VENTILATORULUI

1.11.2.6 (6) – MARCA DE CERTIFICARE CE

1.11.2.7 (7) – INTERFAȚA USB ȘI IDENTIFICAREA ACESTEIA

1.11.2.8 (8) – INTERFAȚA LAN ȘI IDENTIFICAREA ACESTEIA

1.11.2.9 (9) – MARCA DE CERTIFICARE TÜV

1.11.3 MODURI DE OPERARE

- **Mod Independent**: fiecare canal funcționează separat, cu **tensiune** și **curent** ajustabile **individual**;
- **Mod Serie**: **CH1** și **CH2** sunt legate intern, iar **tensiunea totală** este **dublă** celei din canalul individual;
- **Mod Paralel**: **CH1** și **CH2** sunt legate intern pentru a **dubla** curentul de ieșire.

1.11.4 MODUL DE UTILIZARE

În primul rând, se **verifică** integritatea fizică a dispozitivului și accesoriile incluse. Apoi, se **asigură** că selectorul de tensiune de pe panoul din spate **corespunde** rețelei electrice locale, după care se **conectează** cablul de alimentare la sursă și se **pornește** dispozitivul folosind **butonul de alimentare**. Pentru a seta parametrii, se **selectează** canalul dorit utilizând butoanele **CH1**, **CH2** sau **CH3** și se **ajustează** tensiunea sau curentul folosind **knob-ul multifuncțional** și **butonul de ajustare fină**. Pentru a activa ieșirea canalului selectat, se **apasă** butonul **On / Off** și se **monitorizează** parametrii afișați pe ecran pentru verificarea conformității cu valorile dorite. Pentru utilizarea funcțiilor avansate, se **accesează** meniul de afișare a undelor pentru analiza semnalelor, se **configurează** și **utilizează** funcția de **timer** pentru teste automatizate, iar în final se **salvează** și **rețin** setările frecvent utilizate.

1.12 SURSA DE CURENT ALTERNATIV PROGRAMABILĂ ITECH IT7324

Figura 1.21 - Panoul frontal al sursei ITECH IT7324

Caracteristici principale:

- **Putere de ieșire**: **1500 VA**;
- **Tensiune de ieșire**: **150 V (joasă) / 300 V (înaltă)**;
- **Curent de ieșire**: **12 A (joasă) / 6 A (înaltă)**;
- **Gama de frecvență**: **45-500 Hz**;
- **Tehnologie de amplificare liniară**: asigură zgomot redus și stabilitate ridicată;
- **Analizor de putere integrat**: măsoară $\mathbf{V_{rms}}$, $\mathbf{I_{rms}}$, $\mathbf{I_{peak}}$, **frecvența**, **factorul de putere** (**PF**), puterea **activă** și **aparentă**;
- **Simulare perturbări rețea** (***PLD***): permite testarea dispozitivelor în condiții de supratensiune, căderi de tensiune și alte anomalii;
- **Interfețe de comunicare**: dispune standard de interfețe **USB**, **RS-232** și **LAN**.

1.12.1 PANOUL FRONTAL

Panoul frontal este echipat cu un **afișaj clar** și **butoane intuitive** pentru **setarea parametrilor** de ieșire și **navigarea** prin meniuri.

1.12.2 PANOUL POSTERIOR

Panoul posterior include **terminalele de ieșire**, **porturile de comunicare** și **conectorul de alimentare**.

1.12.3 MODUL DE UTILIZARE

Pentru pornire, se **conectează** cablul de alimentare și se **pornește** sursa. Apoi, se **setează tensiunea**, **frecvența** și **alți parametri** doriți utilizând **butoanele** de pe **panoul frontal**. Pentru activarea ieșirii, se **apasă** butonul de activare a ieșirii pentru a alimenta dispozitivul testat, după care se **urmăresc** parametrii măsurați pe afișajul integrat. În cazul în care se dorește control de la distanță, se **utilizează** interfețele de comunicare pentru a controla sursa de la un **PC** cu ajutorul **software-ului dedicat** sau a **comenzilor SCPI**.

1.13 SURSA DE ALIMENTARE AC/DC PROGRAMABILĂ ITECH IT7622

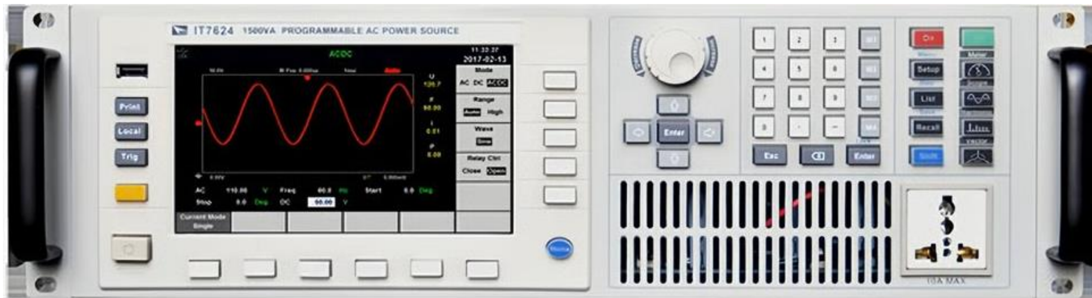


Figura 1.22 - Panoul frontal al sursei ITECH IT7622

Caracteristici principale:

- **Moduri de operare multiple**: poate funcționa în mod **AC**, **DC** și **AC+DC**, permițând simularea distorsiunilor pe o tensiune **DC**;
- **Putere și gamă de ieșire**: oferă o putere maximă de **750 VA**, tensiune de până la **300 Vrms** (**AC**) sau **424 V** (**DC**) și un curent de până la **6 A** (**AC**) sau **3 A** (**DC**);
- **Gamă largă de frecvență**: frecvența de ieșire este ajustabilă de la **10 Hz** la **5 kHz**;
- **Analizor de putere și osciloscop**: include un analizor de putere pentru măsurători detaliate (**Vrms**, **Irms**, **putere activă**, **factor de putere**, etc.) și o funcție de osciloscop pe ecranul de **7 inch** pentru vizualizarea formelor de undă;
- **Generator de forme de undă arbitrare**: permite simularea armonicelor și a altor forme de undă complexe, cu posibilitatea de a importa forme de undă în format **.csv**;
- **Funcții de testare standardizate**: suportă teste conform standardului **IEC 61000-4-11**;
- **Interfețe complete**: echipată standard cu interfețe **USB**, **LAN**, **RS-232** și **CAN**.

1.13.1 PANOUL FRONTAL

Panoul frontal este dominat de un ecran **LCD** mare de **7 inch**, care afișează **formele de undă** și **parametrii** măsurați. Include **butoane de control** pentru **selectarea** modurilor de operare, **ajustarea** setărilor și **navigarea** prin meniuri, precum și terminale de ieșire.

1.13.2 PANOUL POSTERIOR

Pe panoul posterior se găsesc **conectorul de alimentare**, **porturile de comunicație**, **intrările** pentru funcția **"Remote SENSE"** și alți conectori pentru **control** și **sincronizare**.

1.14 SURSA DE ALIMENTARE DC DE MARE PUTERE ITECH IT6522C

Figura 1.23 - Panoul frontal al sursei ITECH IT6522 C

Caracteristici principale:

- **Gamă largă de operare (Auto-Range)**: oferă o putere maximă de **3000 W**, cu o tensiune de până la **80 V** și un curent de până la **120 A**.
- **Operare în două cadrane**: funcționează ca sursă de alimentare și ca sarcină electronică, permițând comutarea rapidă între cele două moduri.
- **Viteză mare de răspuns**: timp de răspuns rapid, cu timpi de creştere şi cădere mai mici de **3 ms**.
- **Moduri de operare versatile**: include moduri de funcționare la **tensiune constantă** (CV), **curent constant** (CC) și **putere constantă** (CP).
- **Funcții de simulare avansate**: dispune de funcție de simulare a curbei **I-V** pentru panouri solare și curbe de tensiune predefinite conform standardelor **DIN 40839** și **ISO-16750-2** pentru rețeaua electrică auto.
- **Paralelizare Master-Slave**: mai multe unități pot fi conectate în **paralel** pentru a obține o capacitate de până la **30 kW**, cu partajarea egală a curentului.
- **Interfețe multiple**: echipată cu interfețe **USB**, **RS232**, **CAN**, **GPIB** și **LAN**, precum și interfețe de control analogic.

1.14.1 PANOUL FRONTAL

Panoul frontal este prevăzut cu un afişaj **VFD** (**Vacuum Fluorescent Display**) pentru vizualizarea clară a setărilor și măsurătorilor, un **knob rotativ** și **butoane pentru configurarea parametrilor** și a **modurilor de operare**, precum și terminalele de ieșire de mare putere.

1.14.2 PANOUL POSTERIOR

Panoul posterior conține **conectorul de alimentare**, **porturile de comunicație** (**USB**, **RS232**, etc.), **conectorii** pentru funcția de detecție la distanță (**Remote Sense**) și **conectorii** pentru controlul analogic.

1.15 BIBLIOGRAFIE

1. *"Power Suply Unit", TEquipment*
2. *"Electronic Load", Rohde & Schwartz*
3. *"Digital Storage Oscilloscope", Siglent*

2. INTRODUCERE ÎN IDE-URILE UTILIZATE

2.1 INTRODUCERE ÎN MPLAB X IDE V6.15

Definiție

MPLAB X IDE (Integrated Development Environment) este un mediu de dezvoltare integrat dezvoltat de Microchip Technology, destinat programării microcontrolerelor din familia PIC (Peripheral Interface Controller) și a altor dispozitive integrate. Versiunea 6.15 a MPLAB X IDE aduce îmbunătățiri semnificative și noi funcționalități care facilitează procesul de dezvoltare a aplicațiilor embedded.

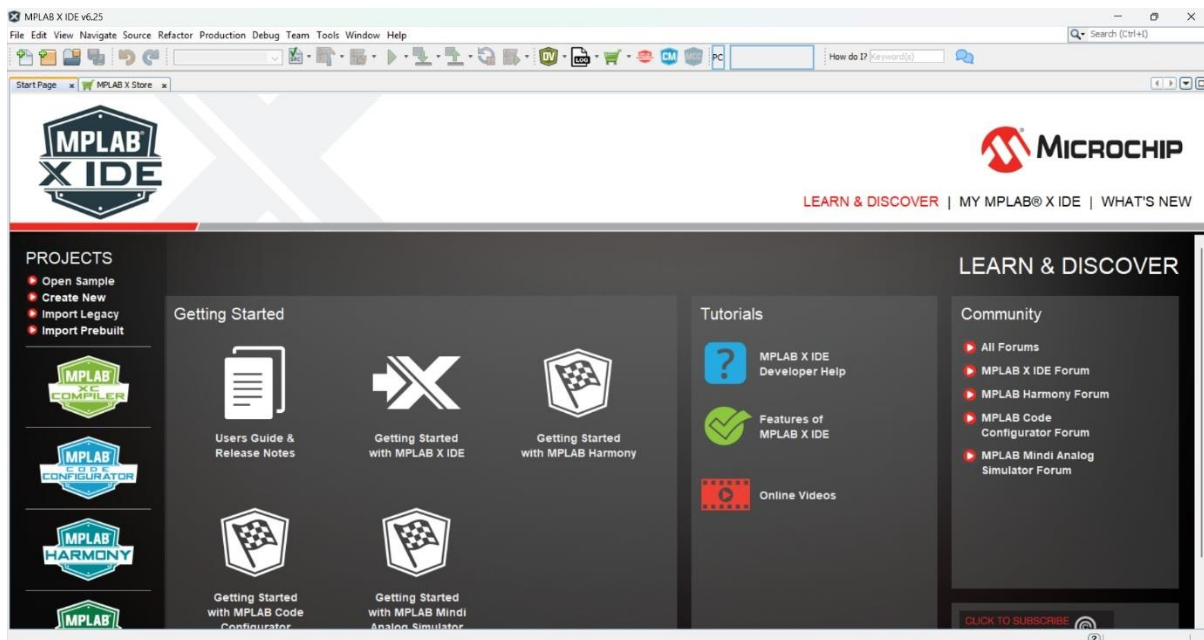


Figura 2.1 - Pagina de start MPLAB X IDE v6.15

2.1.1 NOȚIUNI INTRODUCTIVE DESPRE WORKSPACE

Definiție

Workspace-ul este o componentă fundamentală a oricărui mediu de dezvoltare integrat, inclusiv MPLAB X IDE. Acesta reprezintă un spațiu virtual care permite organizarea proiectelor și a fișierelor asociate.

2.1.2 CARACTERISTICI PRINCIPALE ALE WORKSPACE-ULUI

- **Gestionarea proiectelor multiple:** un **workspace** poate conține **mai multe** proiecte simultan, facilitând comutarea rapidă între ele;
- **Centralizarea resurselor:** toate fișierele **sursă**, **bibliotecile**, și **setările proiectului** sunt organizate logic într-un singur loc;

- **Beneficii:** simplifică **dezvoltarea** și **depanarea** proiectelor complexe, crește productivitatea prin **acces rapid** la **toate fișierele** necesare și **permite configurări diferite** pentru **fiecare proiect** într-un mod flexibil.

Workspace-ul din **MPLAB X IDE** oferă, de asemenea, opțiuni pentru a vizualiza **structura proiectului**, a monitoriza **logurile de compilare** și a configura **debugger-ul**.

2.2 INTRODUCERE ÎN IAR EMBEDDED WORKBENCH 7.30.5

IAR Embedded Workbench este un mediu de dezvoltare integrat (**IDE**) destinat programării sistemelor **embedded**, care include **microcontrolere** și **microprocesoare** din diverse familii, cum ar fi **ARM**, **AVR**, **MSP430** și multe altele. Versiunea **7.30.5** a **IAR Embedded Workbench** vine cu funcționalități avansate care facilitează dezvoltarea de aplicații embedded eficiente și fiabile.

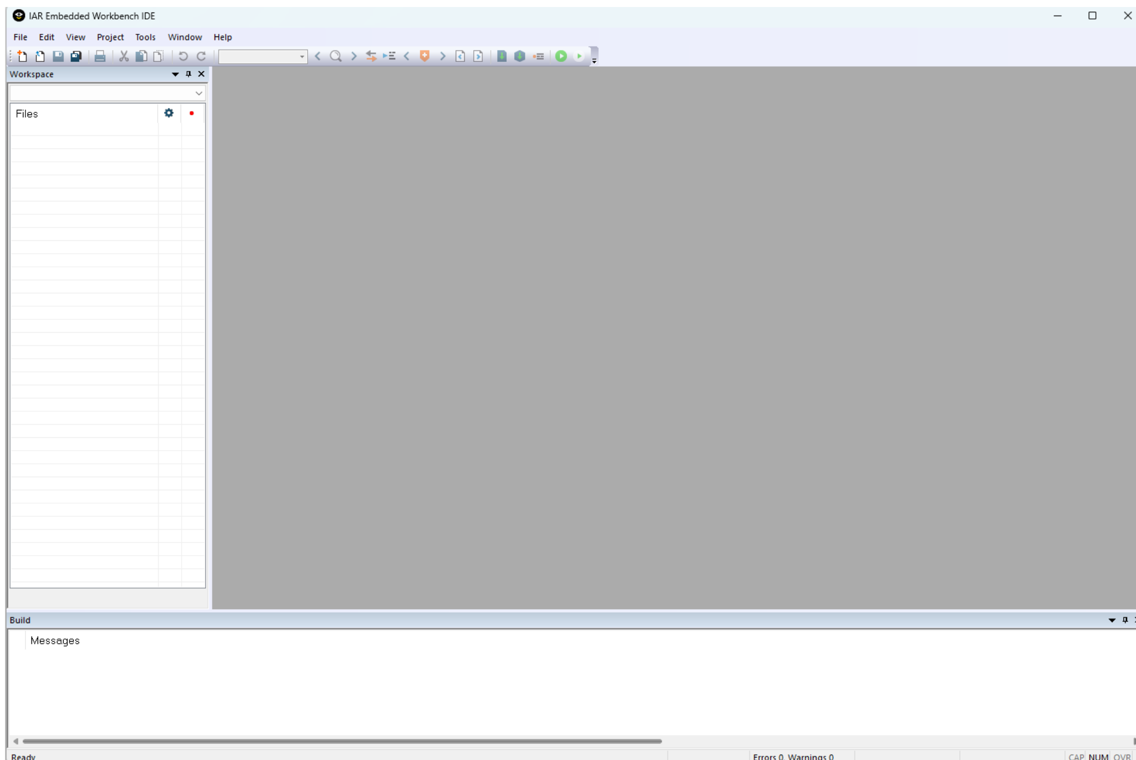


Figura 2.2 - Pagina de start IAR Embedded Workbench v7.30.5

2.2.1 NOȚIUNI INTRODUCTIVE DESPRE WORKSPACE

Workspace-ul din **IAR Embedded Workbench** joacă un rol similar, oferind un mediu **centralizat** pentru gestionarea **fișierelor sursă**, **bibliotecilor** și **setărilor proiectului**.

2.2.2 CARACTERISTICI IMPORTANTE

- **Structura proiectelor: workspace-ul** permite organizarea **logică** a **mai multor proiecte**, fie în cadrul aceluiași **dispozitiv**, fie pentru **dispozitive diferite**.
- **Beneficii:** ușurează **depanarea** și **întreținerea codului**, permite **partajarea configurărilor** între proiecte și reduce timpul de configurare prin opțiuni **predefinite**.

Workspace-ul din **IAR Embedded Workbench** include panouri dedicate pentru **fișiere**, **memorie**, și **variabile**, oferind o privire detaliată asupra proiectului.

2.2.3 CONSTRUIREA DE APLICAŢII – ANSAMBLU

O aplicaţie tipică este construită din **fişiere sursă** şi **biblioteci**. Fişierele sursă pot fi scrise în **C**, **C++** sau **limbaj de asamblare** şi pot fi compilate în fişiere **obiect** de către compilatorul **AVR®IAR** sau **AVR®IAR assembler**.

Linkeditor-ul IAR XLINK este folosit pentru a construi **aplicaţia finală**. **XLINK** foloseşte, în mod normal, un **fişier de comandă** pentru link-editare.

Compilarea: În interfaţa **linei de comandă**, linia următoare compilează fişierul **sursă** **`myfile.c`** în fişierul **obiect** **`myfile.r90`**, folosind setările implicite:

```
iccavr myfile.c.
```

Linkeditarea: **Linkeditor-ul IAR XLINK** este folosit pentru a construi **aplicaţia finală**. În mod normal, **XLINK** necesită următoarele informaţii la intrare:

- Fişiere obiect şi bibliotecile necesare;
- Biblioteca standard ce conţine mediul de rulare şi funcţiile standard ale limbajului;
- Eticheta de start a programului;
- Un fişier de comandă a linkeditorului ce descrie schema memoriei sistemului ţintă;
- Informaţii despre formatul de la ieşire.

În linia de comandă, linia următoare poate fi folosită pentru pornirea XLINK:

```
xlink myfile.r90 myfile2.r90 -s __program_start -f lnkm128s.xcl cl3s-ec.r90
-o aout.a90 -FIntel-extended
```

În acest exemplu , **`myfile.r90`** şi **`myfile2.r90`** reprezintă fişiere **obiect**, **`nkm128s.xcl`** este fişierul de **comandă** al **linkeditorului**, iar **`cl3s-ec.r90`** este **biblioteca de rulare**. Opţiunea **`–s`** specifică locaţia din care aplicaţia porneşte. Opţiunea **`–o`** specifică numele fişierului de ieşire iar opţiunea **`–f`** poate fi folosită pentru a specifica formatul fişierului de ieşire (formatul fişierului de ieşire implicit este Motorola).

Linkeditor-ul IAR XLINK produce ieşirea conform specificaţiilor alese. Formatul de la ieşire se alege conform scopului dorit. Se poate dori încărcarea ieşirii la un **depanator**, ceea ce înseamnă că este nevoie la **ieşire** de informaţii ale depanatorului. Ca alternativă, se poate încerca **ieşirea** la un **flash loader**, caz în care este nevoie de o **ieşire** fără informaţii ale depanatorului cum ar fi **Intel-Hex** sau **Motorola S-Records**.

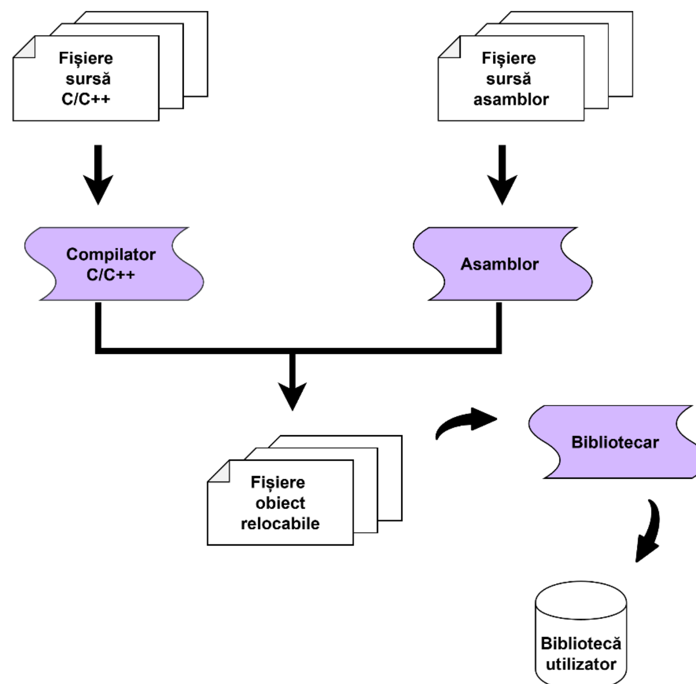


Figura 2.3 – Procesul de translare al codului sursă C în cod sursă în limbaj de asamblare

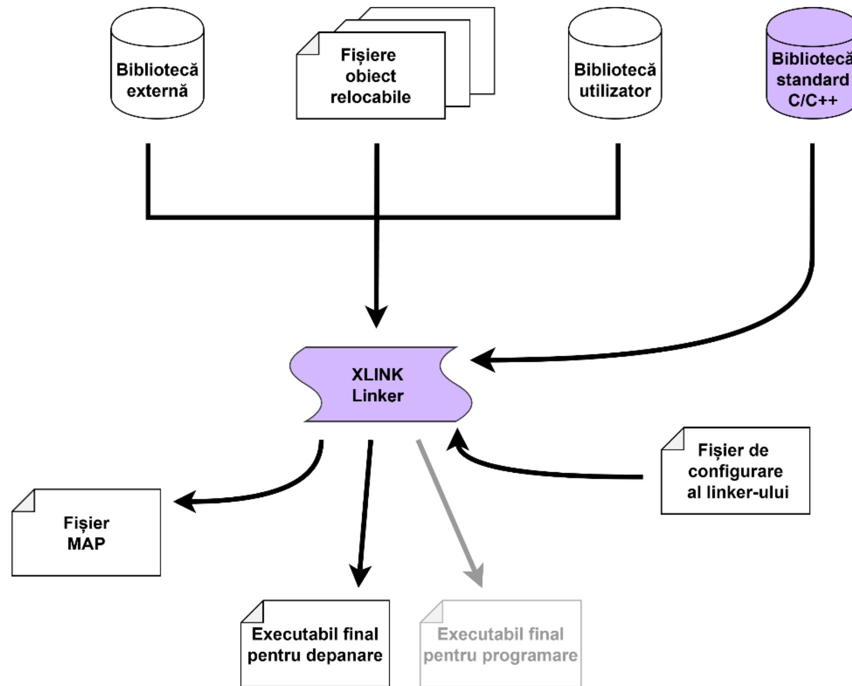


Figura 2.4 – Procesul de link-editare

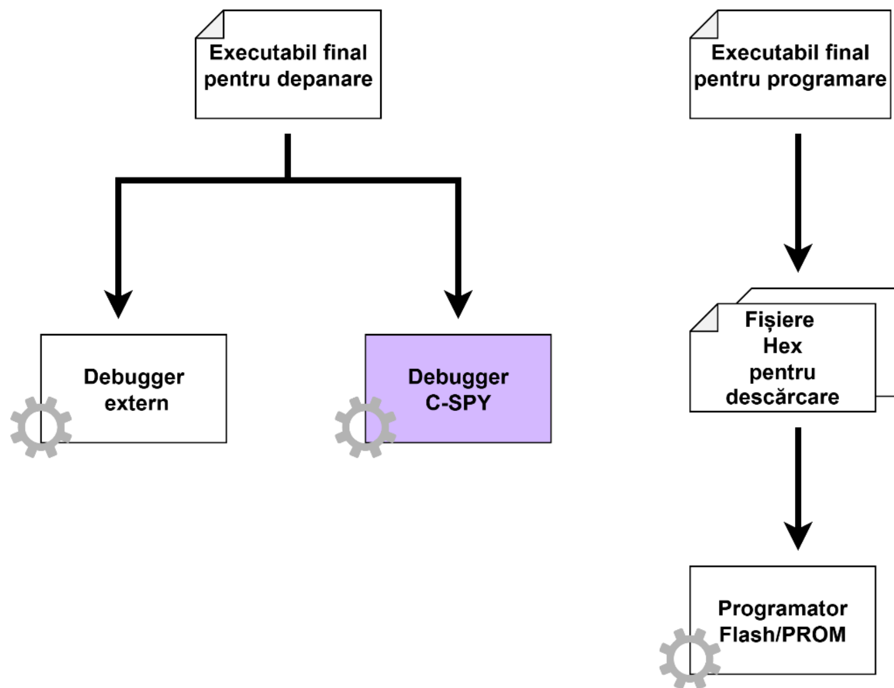


Figura 2.5 – Crearea programului final executabil după link-editare

2.3 REALIZAREA UNUI PROIECT

2.3.1 CREAREA UNUI PROIECT ÎN IAR EMBEDDED WORKBENCH

- se deschide mediul **IAR Embedded Workbench 7.30.5**;
- **menu** = **Project → Create New Project → option** = **Empty Project → button** = **OK**;

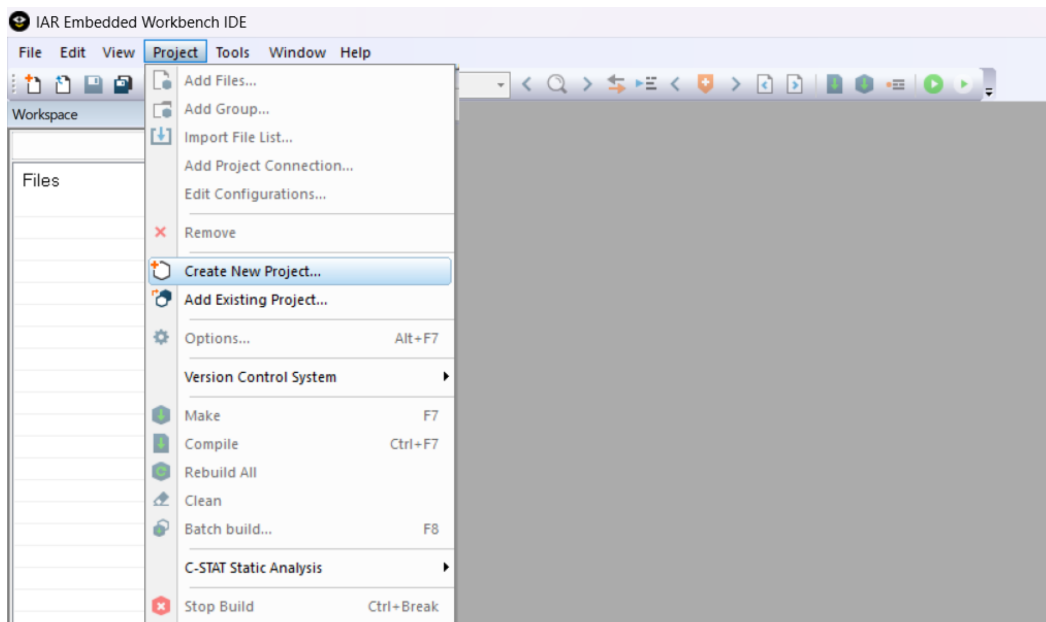


Figura 2.6 – Meniul Project

- se indică **locaţia** şi **numele proiectului** (într-un director nou);

1. **Adăugarea fişierelor sursă:**
   - **menu** = **File** → **menu** = **New** → **menu** = **File**;
   - **menu** = **File** → **menu** = **Save**;
   - **menu** = **Project** → **menu** = **Add files…**

2. **Deschiderea unui proiect existent în IAR**
   - **menu** = **File** → **menu** = **Open Workspace...**

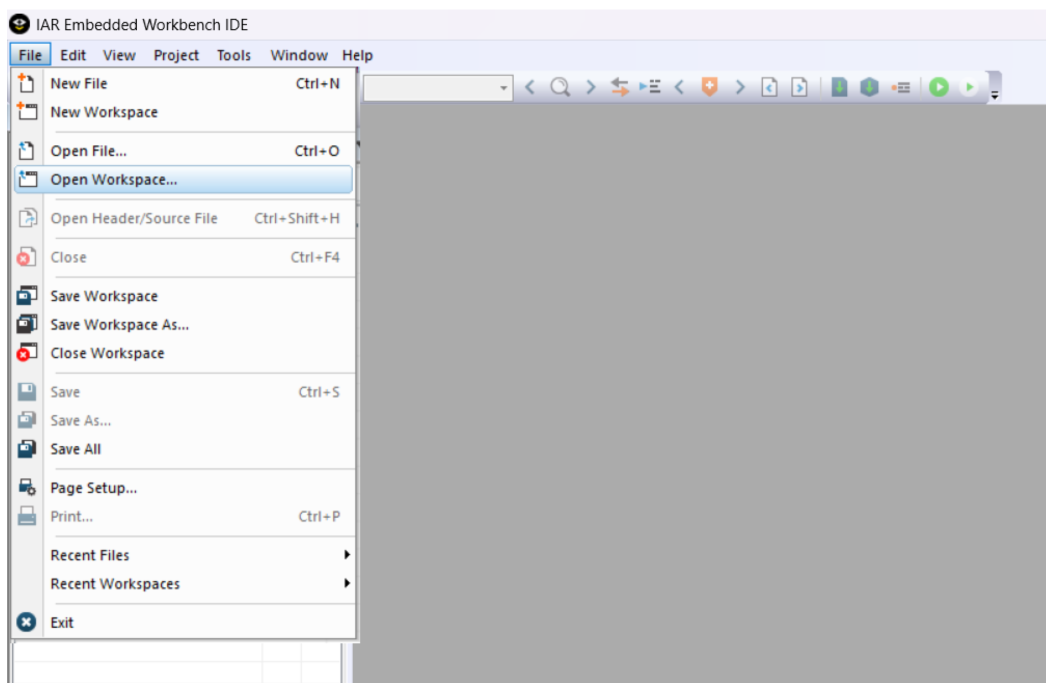


Figura 2.7 – Meniul File

- se indică fişierul **workspace-ului** (cu extensia **.eww**) din directorul corespunzător **proiectului**.

3. **Configurarea proiectului în IAR Embedded Workbench 7.30.5**
   - menu = **Project** ⟶ menu = **Options...**

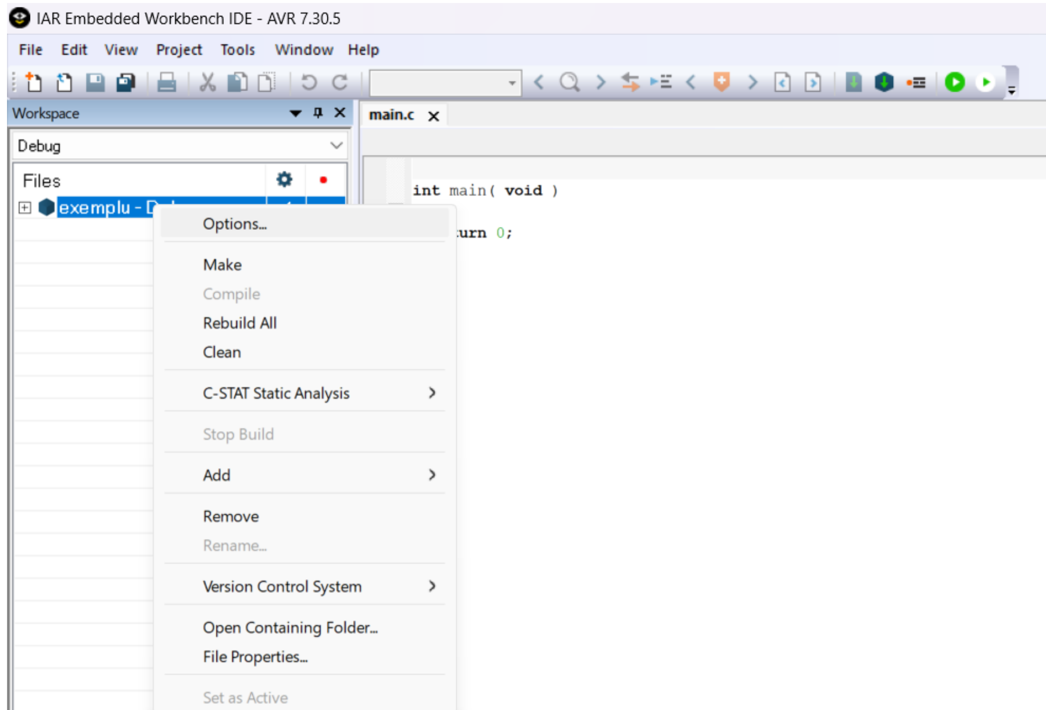


Figura 2.8 - Fereastra workspace-ului

În fereastra ce se va deschide vor fi alese următoarele opțiuni:

- **General Options** ⟶ **Target** ⟶ **Processor Configuration** = **ATmega1280**;

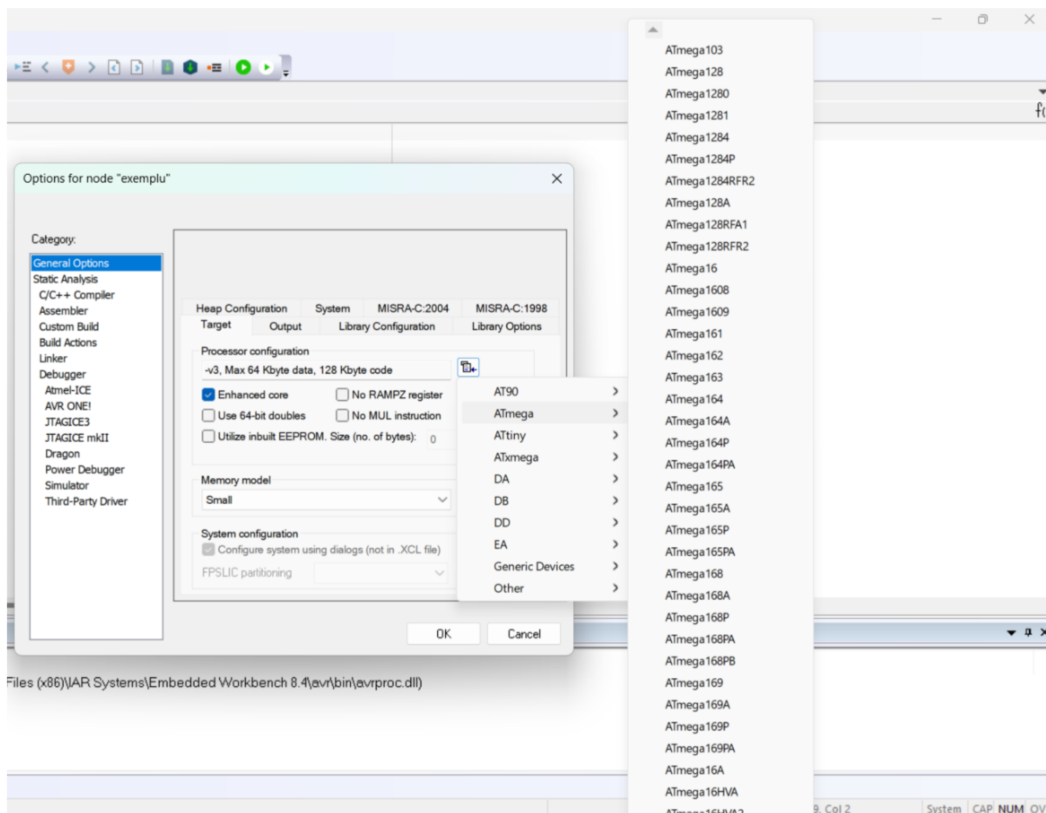


Figura 2.9 - Meniul de selectare al Target-ului

- **General Options → System → Enable bit definitions in I/O-Include files = enabled;**

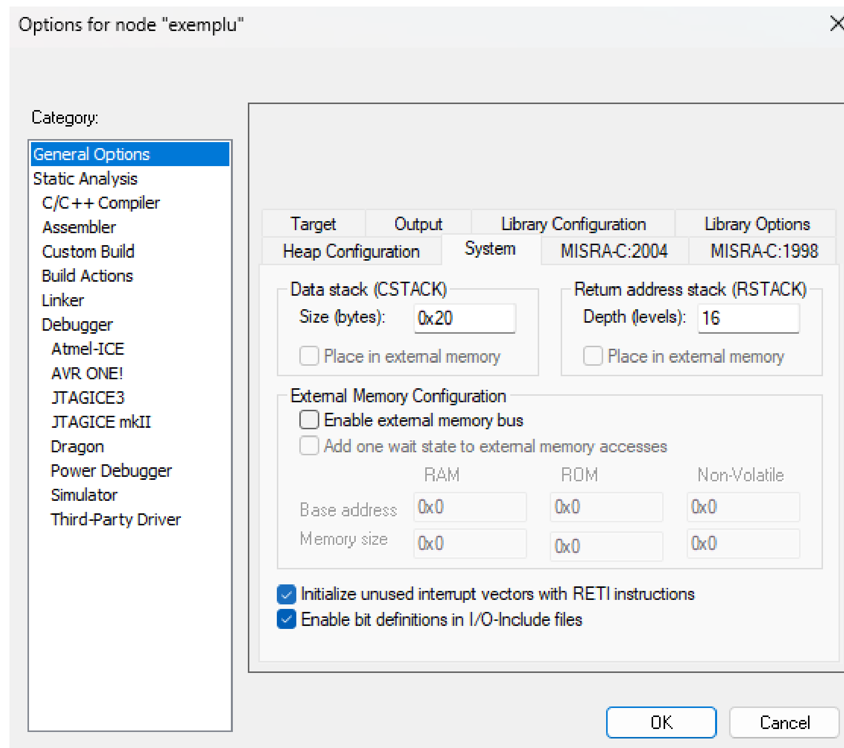


Figura 2.10 - General Options → System

- **C/C++ Compiler → Optimizations = None;**

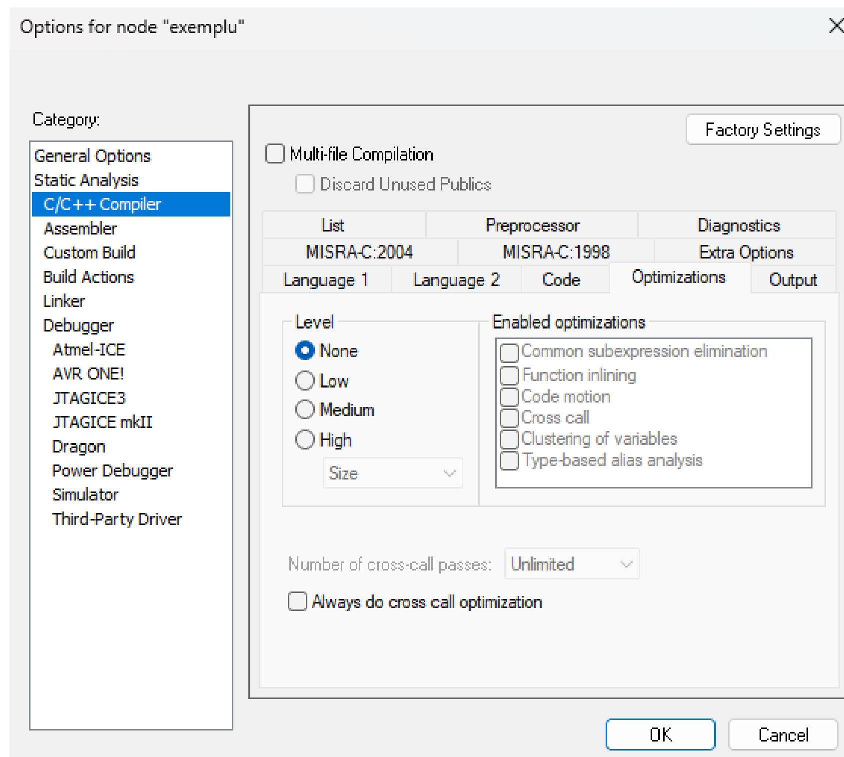


Figura 2.11 - C/C++ Compiler → Optimizations

- **C/C++ Compiler → List → Output list file = enabled** (pentru a obține fișierul **.lst**);

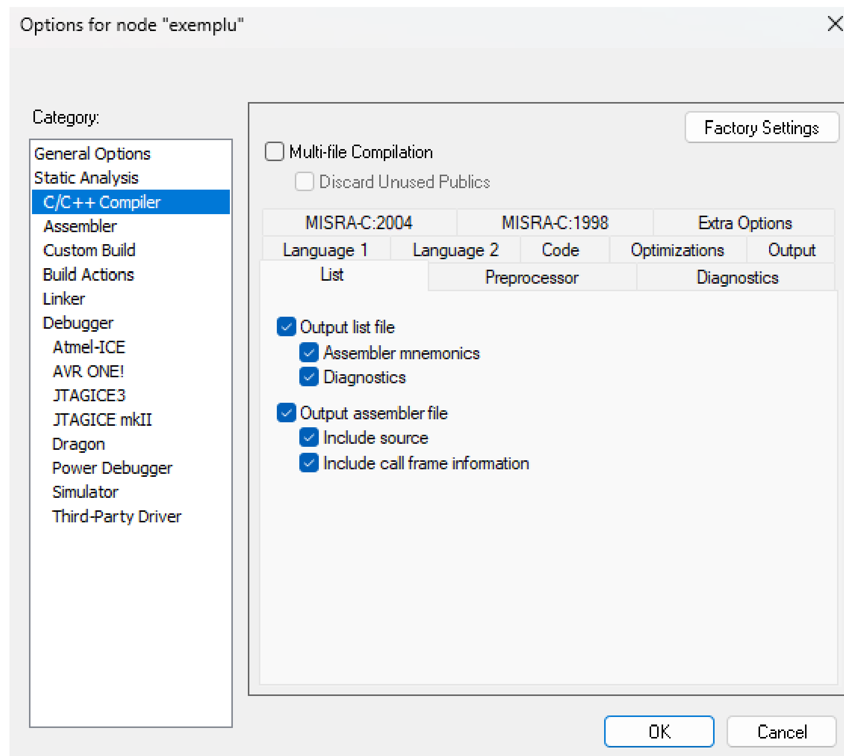


Figura 2.12 - C/C++ Compiler → List

- **Linker → Output → Output Format = C-SPY;**

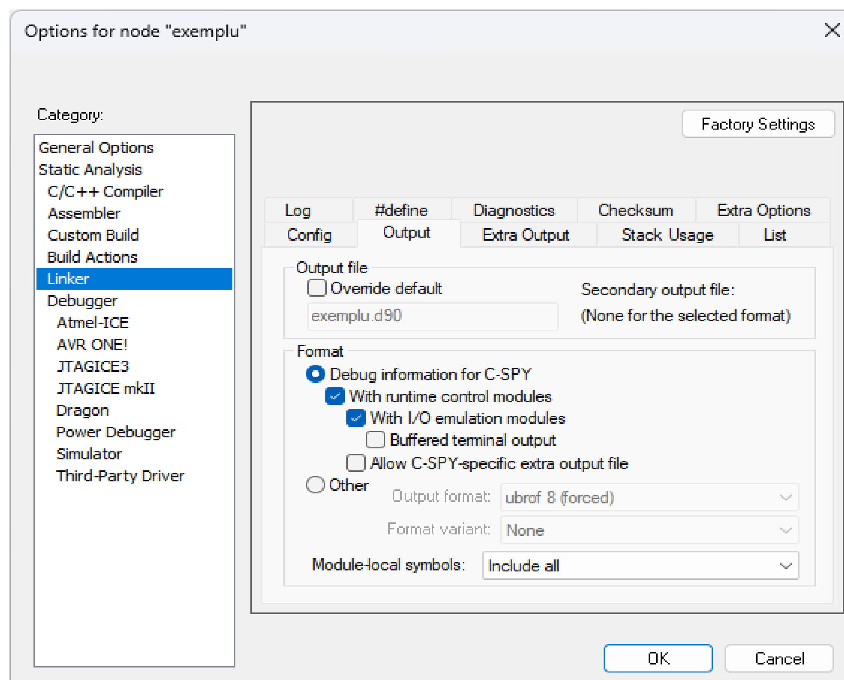


Figura 2.13 - Linker → Output

- **Linker → List → Generate linker listing** = **enabled**, **Segment Map** = **enabled**, **Module map** = **enabled** (pentru a vedea fişierul **.map**), **File format** = **HTML**;

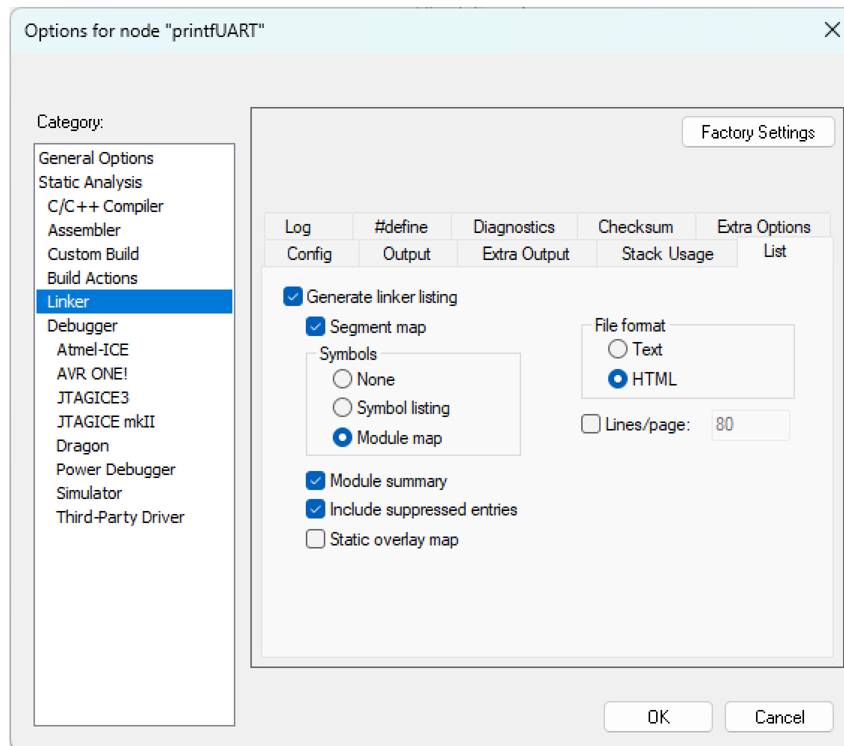


Figura 2.14 - Linker → List

- **Debugger → Setup → Driver → Atmel-ICE**;

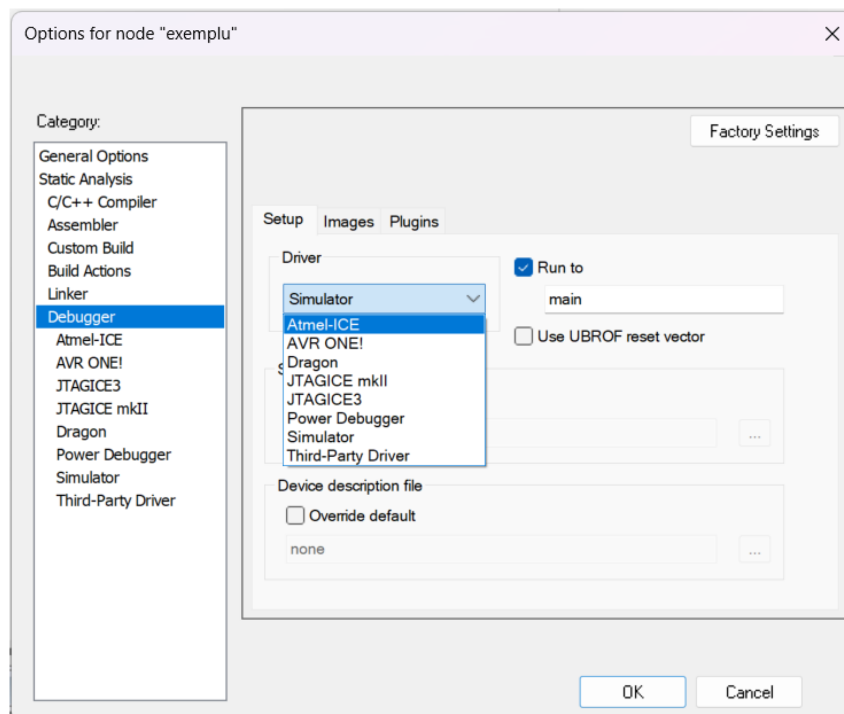


Figura 2.15 - Debugger → Setup → Driver

Înainte de a compila proiectul, se va defini spaţiul de lucru astfel:

- **menu** = **Project** → **menu** = **Make**;
- se va indica locaţia fişierului cu informaţii despre spaţiul de lucru.

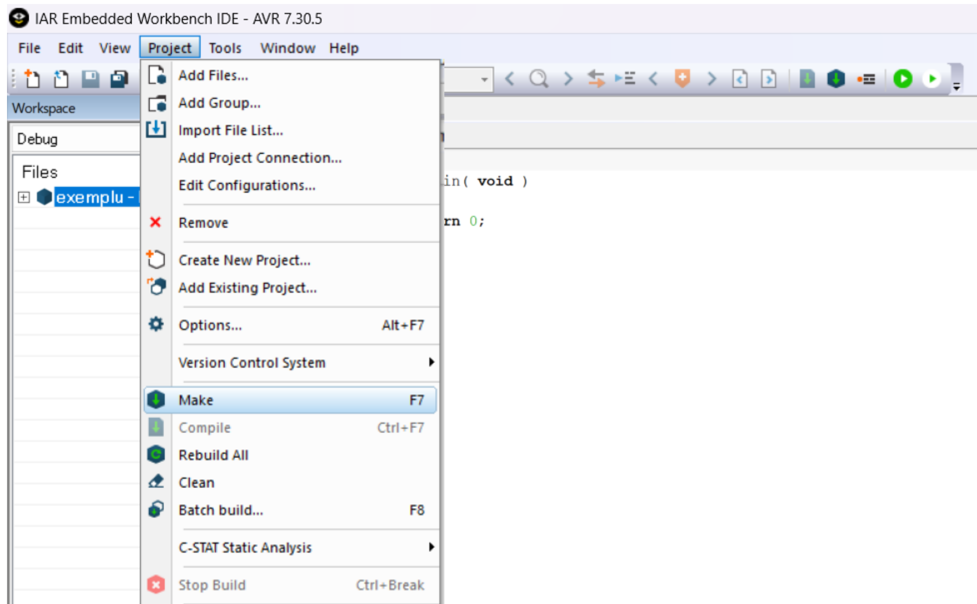


Figura 2.16 - Opțiunea Make din meniul Project

2.3.1.1 PERSPECTIVĂ ASUPRA LIMBAJULUI IAR

Există 2 limbaje de programare de nivel **înalt** disponibile cu compilatorul **AVR®IAR C/C++** :

1. **C**, cel mai **răspândit** limbaj de nivel înalt de programare folosit în industria de **sisteme încorporate**. Folosind compilatorul **AVR®IAR** se pot construi aplicații de sine stătătoare ce urmează standardul **ISO 9899:1990**. Acest standard este cunoscut ca **ANSI C**.
2. **C++**, un limbaj modern **orientat obiect**, cu o **bibliotecă** ce dispune de toate caracteristicile necesare pentru o programare **modulară**. Sistemele **IAR** suportă 2 nivele ale limbajului **C++**:
   a. **Embedded C++** (**EC++**) este un **subset** al standardului de programare **C++** destinat programării **sistemelor încorporate**. Este definit de un consorţiu industrial, **Embedded C++ Technical Comitee**.
   b. **IAR Extended EC++**, cu caracteristici suplimentare cum ar fi suportul total pentru şabloane, suportul pentru spaţiile de nume, operatorii de cast, precum şi **Standard Template Library** (**STL**).

Fiecare din cele 2 limbaje de programare suportate pot fi folosite fie într-un mod **strict**, fie în unul **relaxat**, fie în unul **relaxat** cu **extensiile IAR activate**. Modul strict aderă la **standard**, pe când modul relaxat permite anumite **deviaţii** de la acest standard. Este de asemenea posibil ca anumite părţi ale aplicaţiei să fie implementate în limbaj de asamblare.

2.3.2 CREAREA PROIECTULUI ÎN MPLAB X IDE

- se deschide mediul **MPLAB X IDE v6.15**;
- **menu** = **File → New Project...**

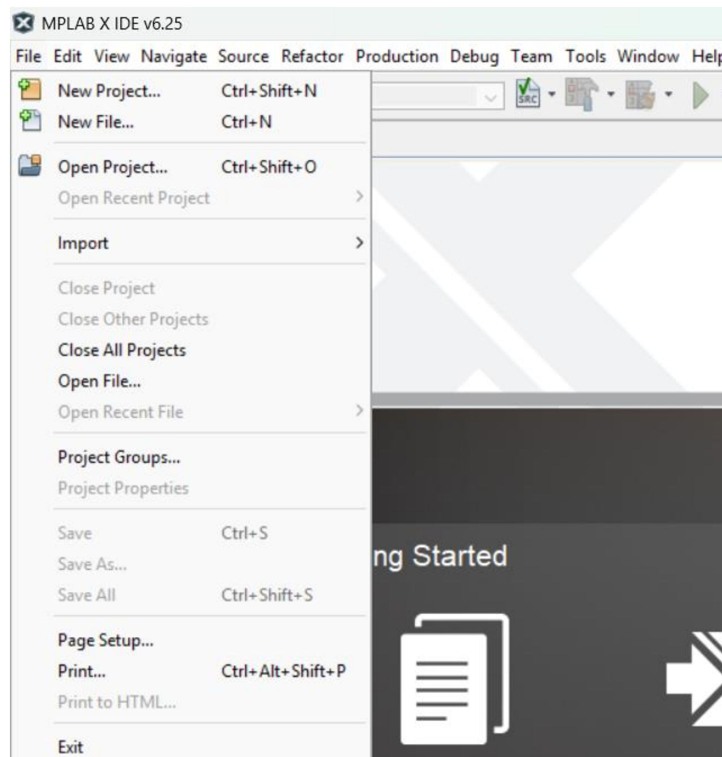


Figura 2.17 - Meniul File

- **menu** = **Categories** → **option** = **Microchip Embedded** → **menu** = **Projects** → **option** = **Standalone Project** → **button** = **Next**;

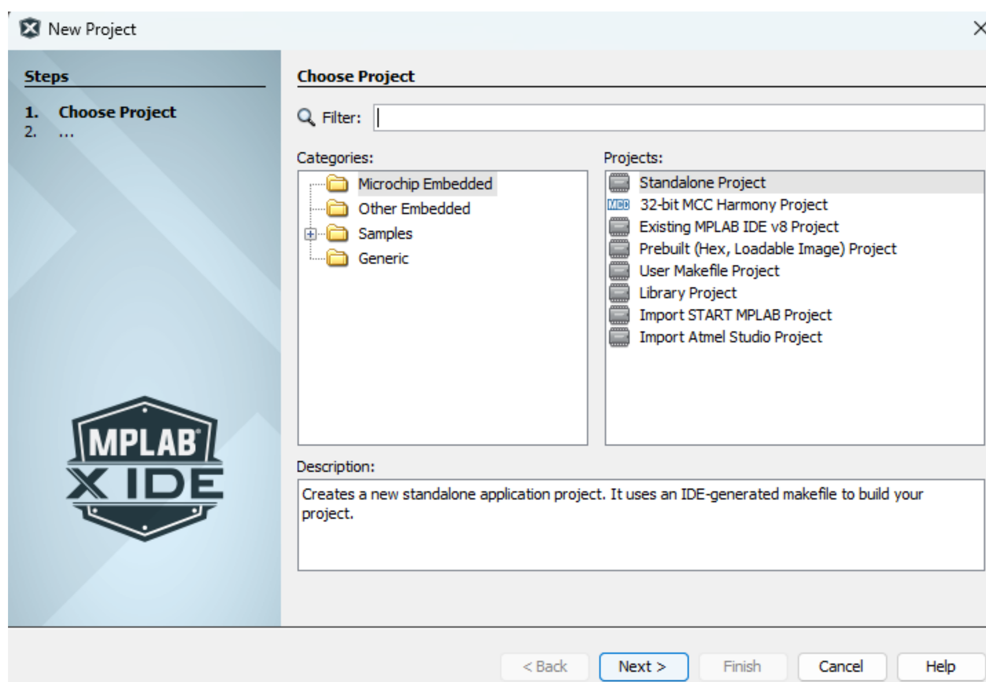


Figura 2.18 - Meniul de selectare al tipului de proiect

- **menu** = **Family** → **option** = **8-bit AVR MCU (Xmega / Mega / Tiny / AVR)** → **menu** = **Device** → **option** = **ATmega1280** → **menu** = **Tool** → **option** = **Atmel-ICE** → **button** = **Next**;

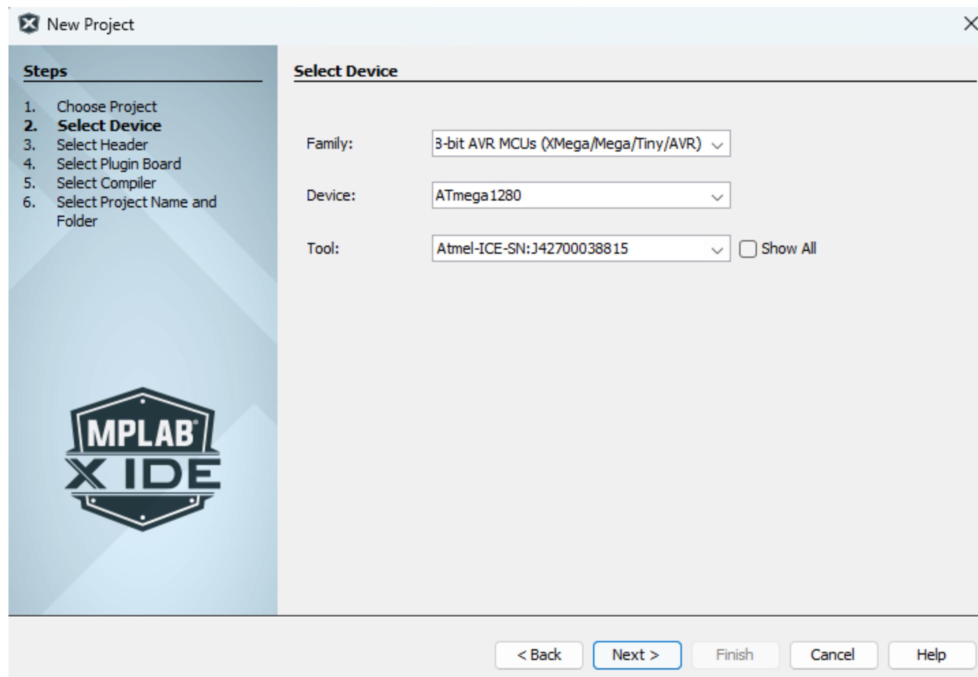


Figura 2.19 - Meniul de selectare al device-ului

- **menu** = **Compiler Toolchains** → **option** = **XC8** → **button** = **Next**;

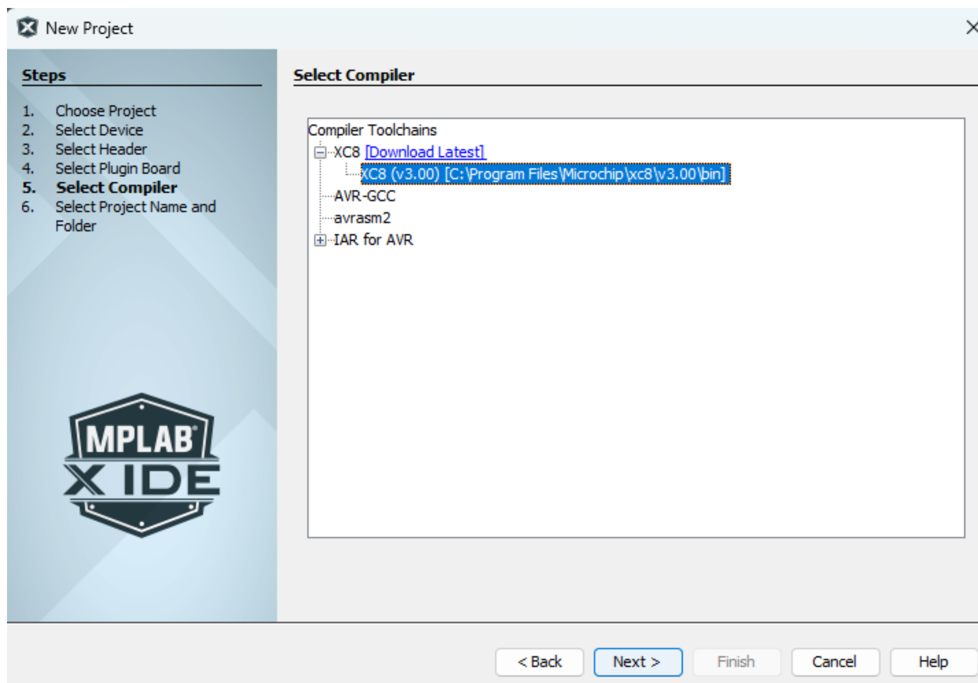


Figura 2.20 - Meniul de selectare al compilatorului

- **menu** = **Project Name** → **numeProiect** → **menu** = **Project Location** → **folderPersonal** → **option** = **Set as main project** → **menu** = **Encoding** → **option** = **UTF-8** → **button** = **Next**;

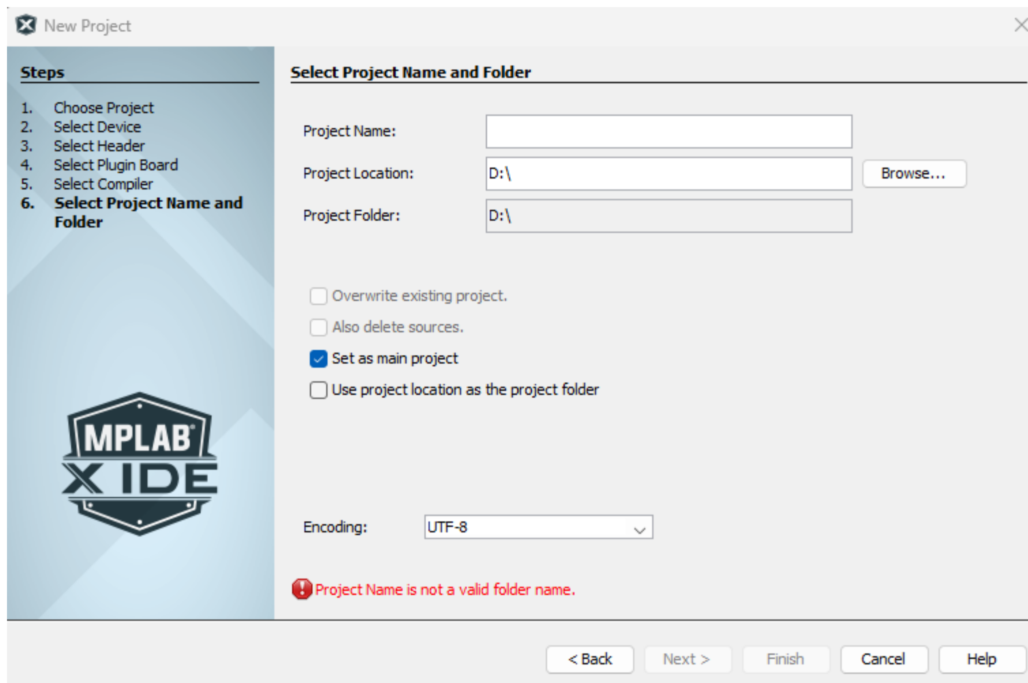


Figura 2.21 - Meniul de selectare al numelui și căii proiectului

Adăugarea fișierelor sursă:

- **menu** = **File** → **option** = **New File...**
- **menu** = **Categories** → **option** = **Limbajul dorit** → **menu** = **File Type** → **option** = **Fișierul dorit**;
- **menu** = **Name and Location** → **option** = **File Name** → **button** = **Finish**.

1. Deschiderea unui proiect existent în MPLAB:

- **menu** = **File** → **option** = **Open Project**;
- se indică fișierul corespunzător proiectului (cu extensia **.X**).

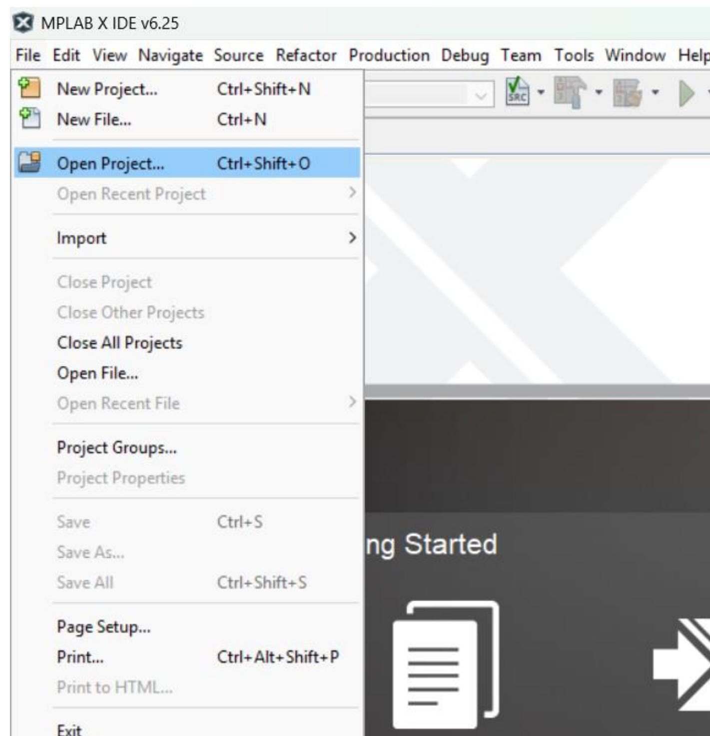


Figura 2.22 - Opțiunea Open Project din meniul File

2. Configurarea proiectului în MPLAB:

- **menu** = **Production** → **menu** = **Set Project Configuration** → **option** = **Customize...**

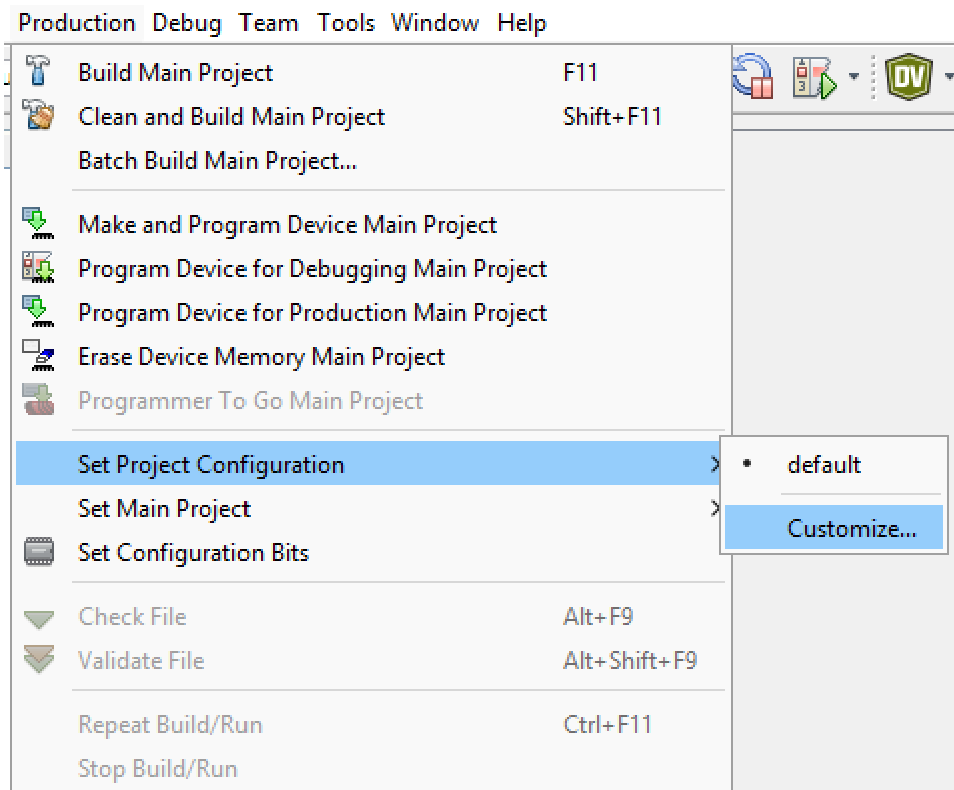


Figura 2.23 - Opțiunea Set Project Configuration din meniul Production

- **menu** = **Conf** → **menu** = **Device** → **option** = **ATmega1280**;
- **menu** = **Connected Hardware Tool** → **option** = **Atmel-ICE**;
- **menu** = **Packs** → **option** = **ATmega_DFP** (doar în simulator);
- **menu** = **Compiler Toolchains** → **option** = **XC8**.

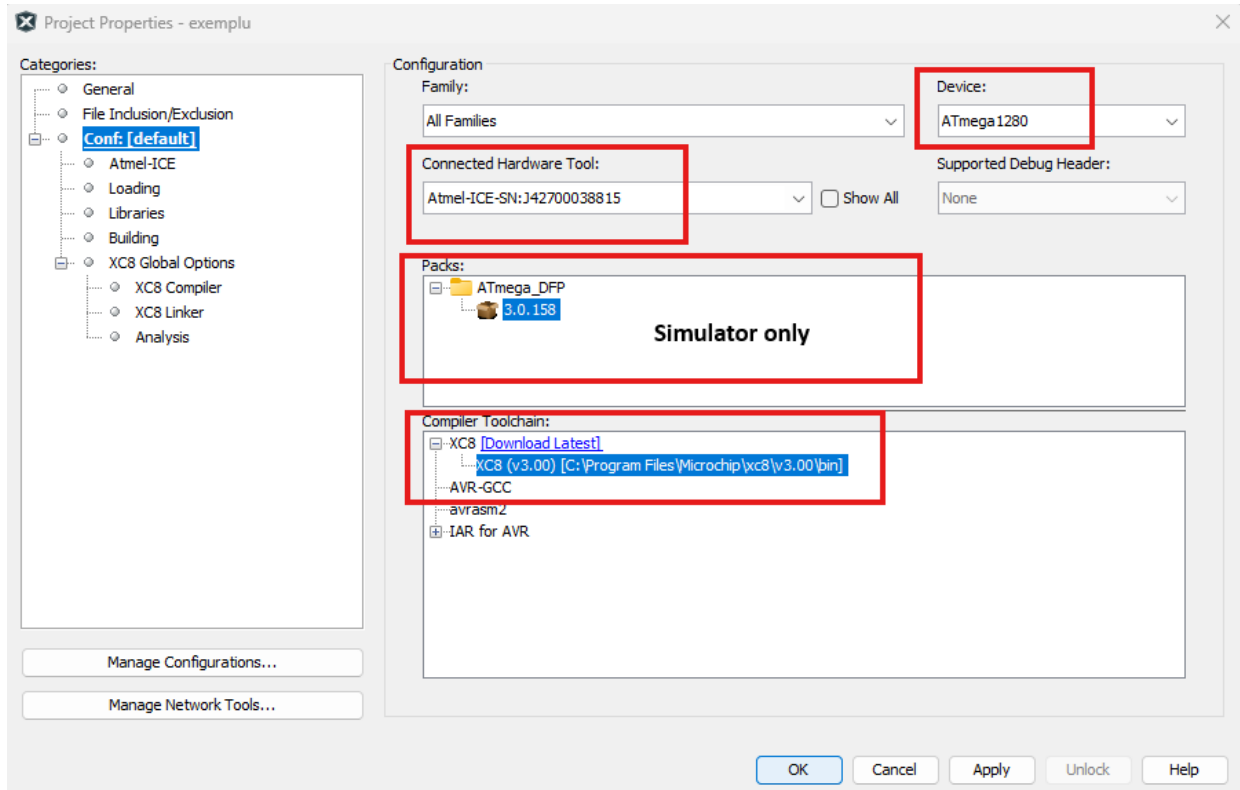


Figura 2.24 - Fereastra de proprietăți ale proiectului

- **menu = Conf → menu = Atmel-ICE → menu = Option categories → Memories To Program;**
- **Auto select memories and ranges → Manually select memories and ranges;**
- **option = Configuration Memory → unchecked;**
- **option = Preserve Data Flash → checked.**

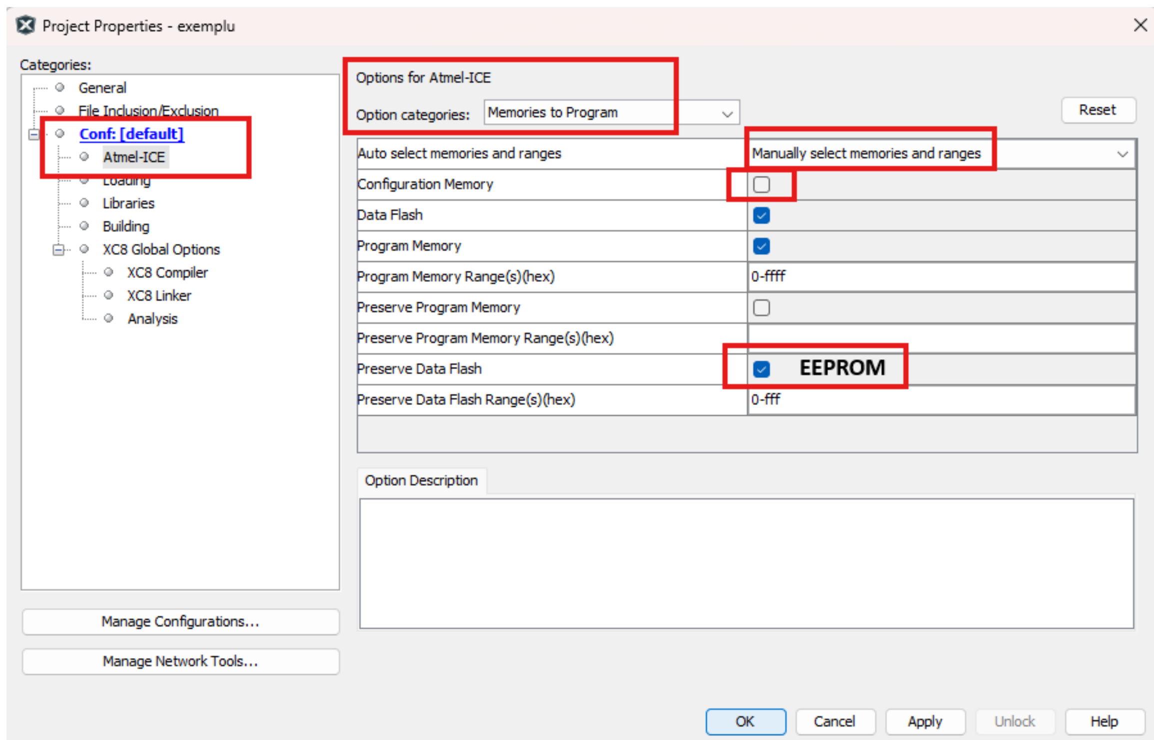


Figura 2.25 - Fereastra Memories To Program

- **menu** = **Conf** → **menu** = **Atmel-ICE** → **menu** = **Option categories** → **option** = **Debug Options**;
- **menu** = **Debug startup** → **option** = **Half at Reset Vector**;
- **menu** = **Debug reset** → **option** = **Reset Vector**.

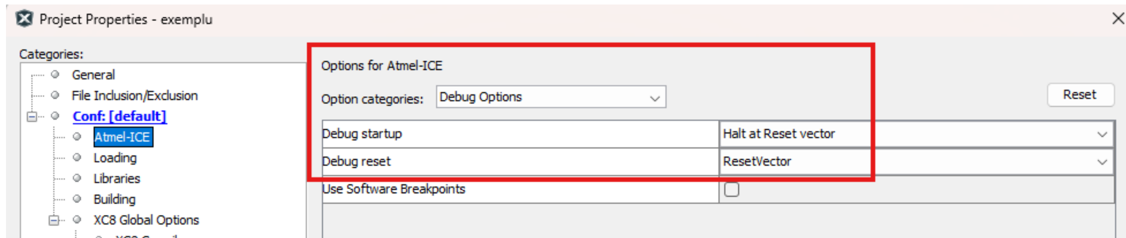


Figura 2.26 - Fereastra Debug Options

- **menu** = **Conf** → **menu** = **Atmel-ICE** → **menu** = **Option categories**;
- **menu** = **Interface** → **option** = **JTAG**;
- **menu** = **Speed** (**MHz**) → **option** = **1**.

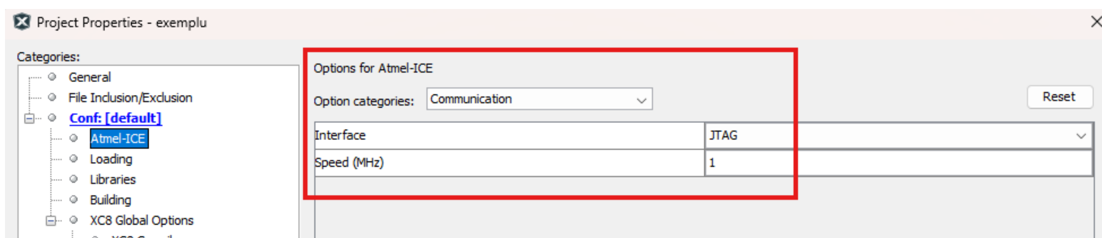


Figura 2.27 - Fereastra Communication

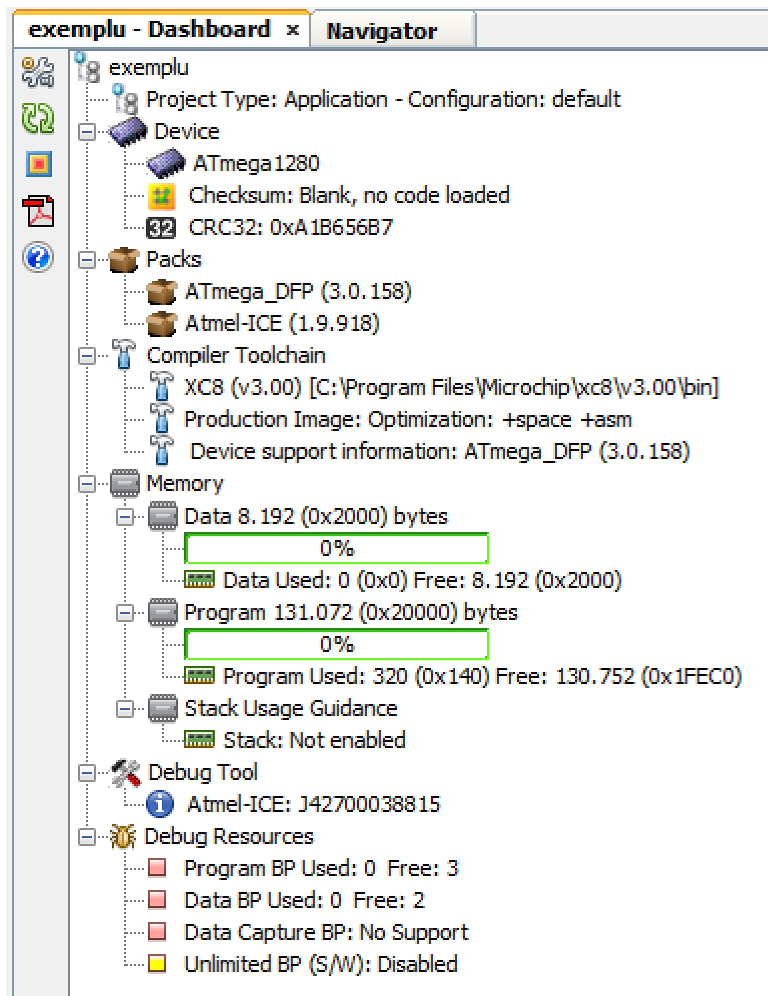


Figura 2.28 - Dashboard-ul proiectului

2.4 DEPANAREA UNUI PROIECT

Definiție

Depanarea (debugging) reprezintă o etapă esențială în dezvoltarea aplicațiilor embedded, ajutând la identificarea și rezolvarea erorilor din cod, pentru a obține un sistem funcțional și optimizat.

IAR Embedded Workbench și MPLAB X IDE sunt două dintre cele mai folosite medii de dezvoltare integrată pentru microcontrolere, fiecare oferind unelte specializate pentru depanare și testare.

Pentru o înțelegere detaliată a procesului, în continuare sunt descriși pașii necesari pentru **configurarea** și **depanarea** unui proiect în **IAR Embedded Workbench**. Fiecare etapă este însoțită de capturi de ecran care ilustrează acțiunile necesare și setările relevante. Urmând acești pași, utilizatorii pot accesa opțiunile de **debugging**, asigurându-se că toate componentele sunt **corect integrate** și **funcționale**.

2.4.1 DEPANAREA UNUI PROIECT ÎN IAR EMBEDDED WORKBENCH

În **IAR Embedded Workbench**, procesul de **debugging** permite dezvoltatorilor să analizeze codul **pas cu pas** și să folosească funcții avansate de monitorizare a **variabilelor** și a **regiștrilor**.

IAR oferă suport pentru limbajele **C** și **C++** (inclusiv subsetul **Embedded C++**), și integrează tool-uri puternice precum **IAR XLINK** pentru **managementul memoriei** și **generarea fișierelor de ieșire**, simplificând procesul de compilare și link-editare.

Pentru a facilita procesul de depanare în **IAR Embedded Workbench**, este esențial să parcurgem câțiva pași esențiali care permit o analiză detaliată și control asupra execuției codului. Pașii următori descriu cum setăm puncte de oprire (**breakpoints**) și monitorizăm **variabilele**, precum și cum folosim ferestrele și instrumentele disponibile pentru a vizualiza în profunzime funcționarea aplicației.

Pașii de depanare includ:

1. **Make** – Compilarea proiectului pentru a verifica dacă nu există **erori de sintaxă** și pentru a pregăti aplicația pentru **depanare**.

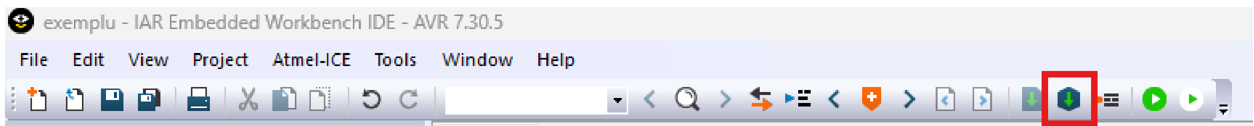


Figura 2.29 - Butonul Make

2. **Setarea unui breakpoint** – Stabilirea punctelor de **oprire** pentru a putea **analiza** anumite părți ale codului.

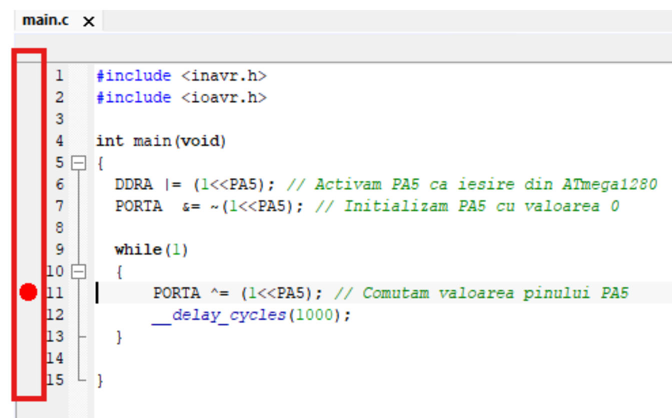


Figura 2.30 - Setarea unui breakpoint

3. **Download And Debug** – **Încărcarea** aplicației pe **microcontroler** și **inițierea** sesiunii de **debugging**.

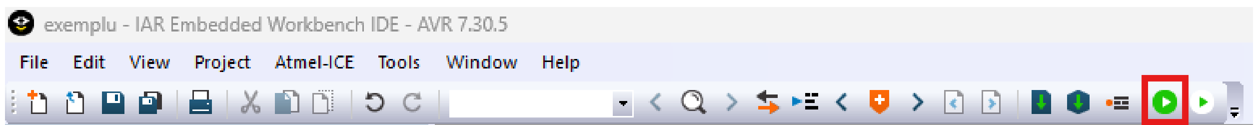


Figura 2.31 - Butonul Download And Debug

4. **Adăugarea unui watch pentru variabile (doar în modul DEBUG)** – Monitorizarea variabilelor de interes pentru a urmări **modificările** acestora în **timpul execuției**.

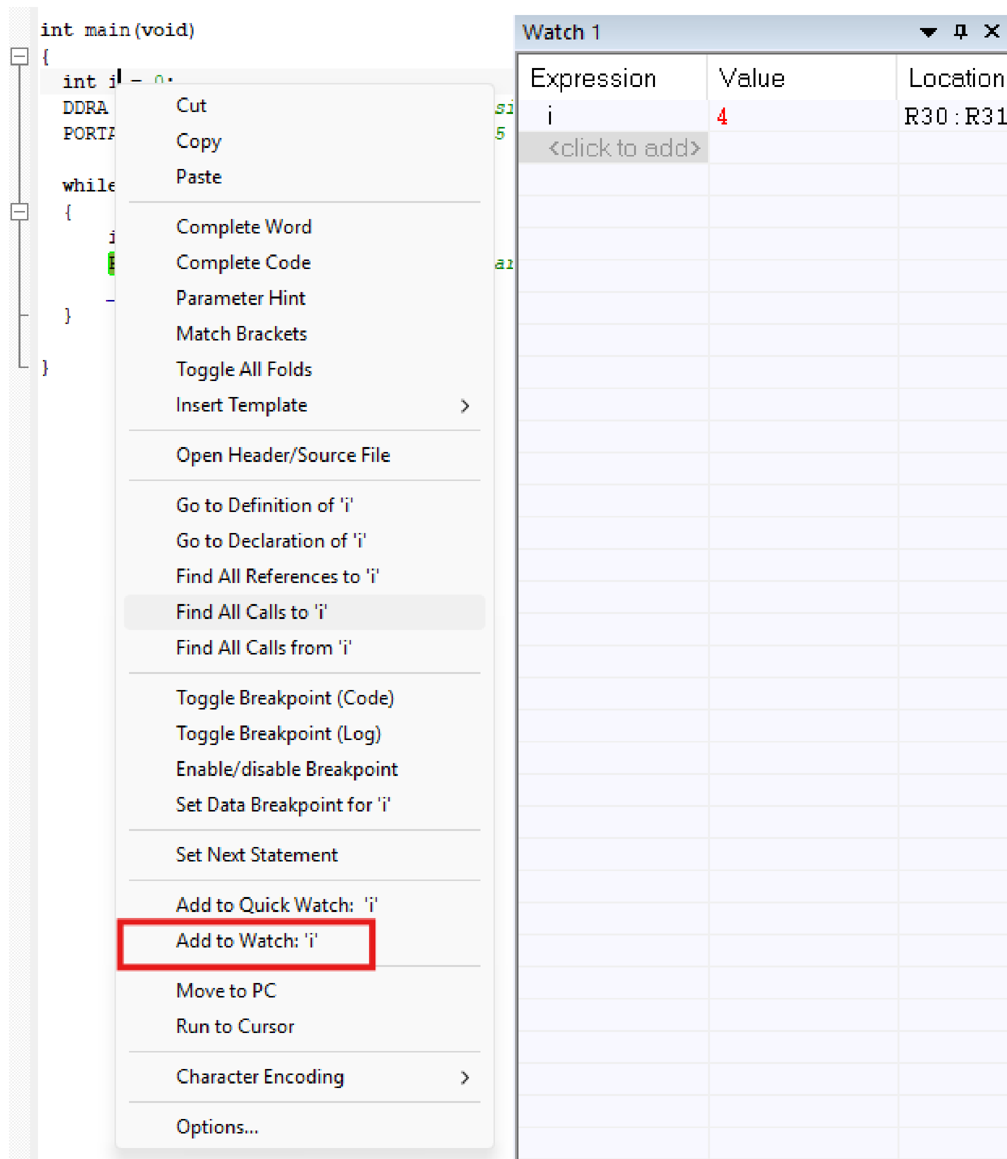


Figura 2.32 - Meniul Watch

5. **Go (până la breakpoint)** – Rularea codului până la următorul **punct de oprire** definit.

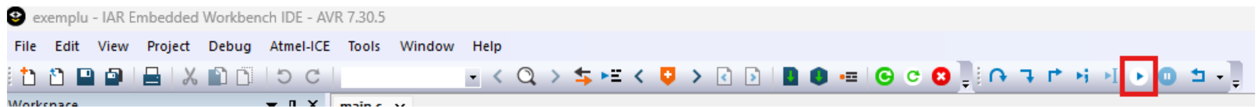


Figura 2.33 - Butonul Go

6. **Break** – Suspendarea execuției pentru a analiza starea aplicației într-un moment intermediar.

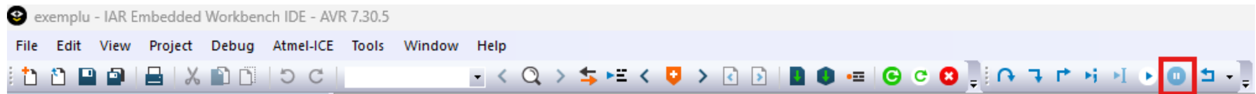


Figura 2.34 - Butonul Break

7. **Ferestre disponibile în timpul procesului de depanare** – Accesarea **ferestrelor de debugging** oferite de **IAR Embedded Workbench** pentru un control mai amănunțit asupra execuției.

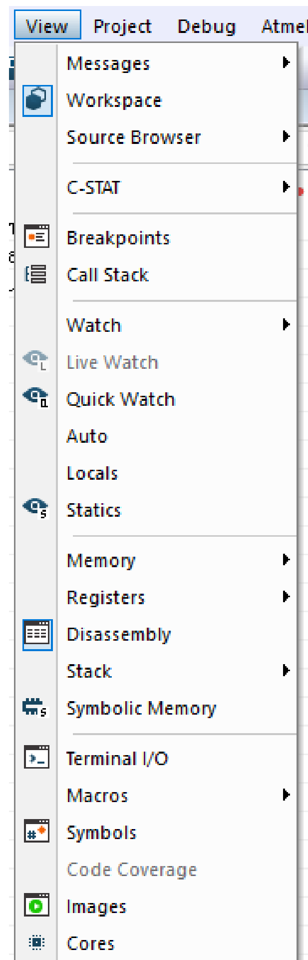


Figura 2.35 - Meniul View

8. **Vizualizarea regiștrilor** – Examinarea **regiștrilor procesorului** pentru a analiza detalii despre starea **internă** a microcontrolerului.

Figura 2.36 - Vizualizarea regiștrilor

9. **Disassembly** – Vizualizarea codului la **nivel de asamblare** pentru o analiză detaliată.

```
Disassembly
Go to                      CODE
Disassembly
  0000A8   9518          RETI
  0000AA   9518          RETI
  0000AC   9518          RETI
  0000AE   9518          RETI
  0000B0   9518          RETI
  0000B2   9518          RETI
  0000B4   9518          RETI
  0000B6   9518          RETI
  0000B8   9518          RETI
  0000BA   9518          RETI
  0000BC   9518          RETI
  0000BE   9518          RETI
  0000C0   9518          RETI
  0000C2   9518          RETI
  0000C4   9518          RETI
  0000C6   9518          RETI
  0000C8   9518          RETI
  0000CA   9518          RETI
  0000CC   9518          RETI
  0000CE   9518          RETI
  0000D0   9518          RETI
  0000D2   9518          RETI
  0000D4   9518          RETI
  0000D6   9518          RETI
  0000D8   9518          RETI
  0000DA   9518          RETI
  0000DC   9518          RETI
  0000DE   9518          RETI
  0000E0   9518          RETI
  0000E2   9518          RETI
  int i = 0;
main:
  0000E4   E0E0          LDI     R30,0x00
  0000E6   E0F0          LDI     R31,0x00
  DDRA |= (1<<PA5); // Activam PA5 ca iesire din ATmega1280
  0000E8   9A0D          SBI     0x1,5
  PORTA  &= ~(1<<PA5); // Initializam PA5 cu valoarea 0
  0000EA   9815          CBI     0x2,5
      i++;
  0000EC   9631          ADIW    R30,1
      PORTA ^= (1<<PA5); // Comutam valoarea pinului PA5
  0000EE   B112          IN      R17,PORTA
  0000F0   E200          LDI     R16,0x20
  0000F2   2710          EOR     R17,R16
  0000F4   B912          OUT     PORTA,R17
      __delay_cycles(1000);
  0000F6   EF0A          LDI     R16,0xFA
  0000F8   0000          NOP
  0000FA   950A          DEC     R16
  0000FC   F7E9          BRNE    0xF8
  0000FE   CFF6          RJMP    0x0EC
__DebugBreak:
  000100   9508          RET
__exit:
```

Figura 2.37 - Fereastra Disassembly

10. **Step By Step** – Execuția **pas cu pas** a codului pentru o înțelegere precisă a fiecărei instrucțiuni.

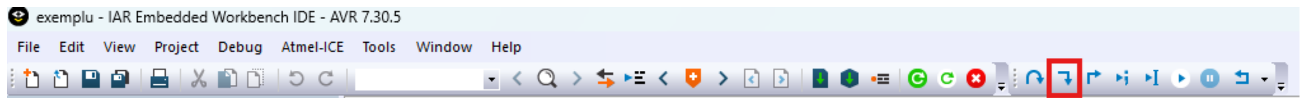


Figura 2.38 - Butonul Step By Step

11. **Stop Debugging** – **Oprirea** procesului de **debugging** după finalizarea analizei.

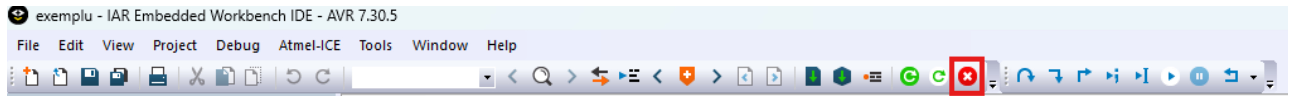


Figura 2.39 - Butonul Stop Debugging

2.5 CREAREA UNEI BIBLIOTECI ÎN PROIECTELE EMBEDDED

În cadrul dezvoltării aplicațiilor **embedded**, **bibliotecile** reprezintă un **set de funcționalități reutilizabile**, organizate în **module de cod** care pot fi ușor incluse în diverse proiecte. Crearea unei biblioteci asigură **modularitatea**, simplifică **gestionarea codului** și promovează **reutilizarea funcțiilor testate și optimizate**. În această secțiune, vom descrie procesul de creare a unei biblioteci în **IAR Embedded Workbench**, abordând configurarea proiectului, generarea **fișierelor de ieșire** și includerea bibliotecii în alte proiecte.

2.5.1 CE ESTE O BIBLIOTECĂ?

Definiție

O bibliotecă este o colecție de funcții și resurse care pot fi utilizate în mai multe aplicații fără a fi necesară reimplementarea acestora.

Avantajele folosirii unei biblioteci sunt obținerea unui **cod modular**, creșterea **reutilizării aceluiași cod**, creșterea **ușurinței în mentenanță**, **creșterea performanței** și **reducerea dimensiunii codului sursă** în proiectele ce utilizează aceleași funcționalități.

Există biblioteci **statice**, unde codul este inclus **direct** în timpul compilării și biblioteci **dinamice** (acolo unde sunt permise), unde codul este **încărcat în timpul rulării**, dar în proiectele **embedded** acest lucru este mai **rar întâlnit** din cauza **limitărilor** de **resurse** și a **cerințelor de performanță**.

2.5.2 CREAREA UNEI BIBLIOTECI ÎN IAR EMBEDDED WORKBENCH

În **IAR Embedded Workbench**, crearea unei biblioteci permite dezvoltatorilor să **grupeze funcționalități comune** într-un fișier **`.r90`**, care poate fi reutilizat în alte proiecte.

- **Project Explorer → Options → General Options → Output → Output File → Library**.

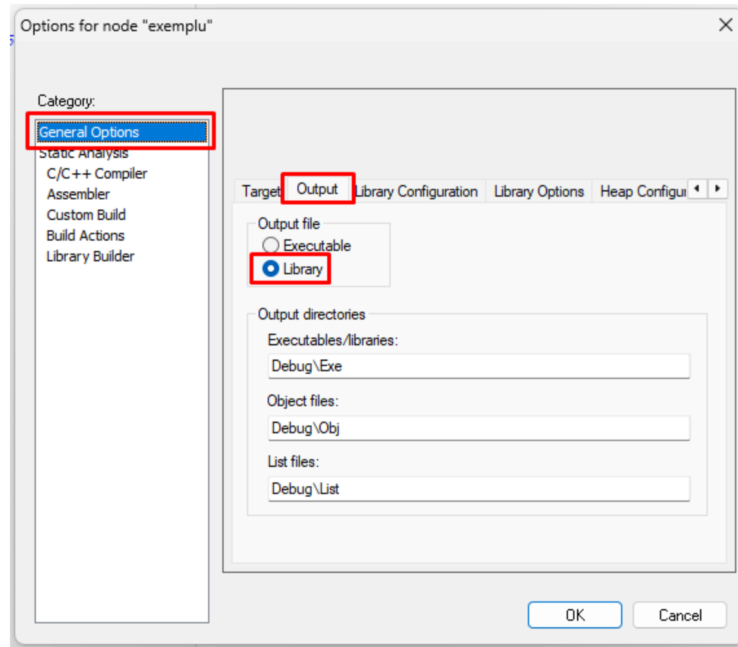


Figura 2.40 - General Options → Output

- După ce opțiunea de ieșire a fost setată la **Library**, **compilați proiectul** folosind comanda **Make**.
- Biblioteca va fi generată în directorul de build specificat, de obicei în format `.r90`, și va fi localizată în **proiect\Debug\Obj\nume_librarie.r90**.

2.6 INCLUDEREA UNEI BIBLIOTECI ÎN PROIECTELE EMBEDDED

Includerea bibliotecilor este o practică **esențială** în dezvoltarea de proiecte **embedded**, deoarece permite **reutilizarea** unor **module de cod precompilate**, contribuind astfel la **eficiența dezvoltării** și la **modularitatea proiectului**. În această secțiune, vom descrie pașii necesari pentru a include o bibliotecă în proiectele dezvoltate în **IAR Embedded Workbench**.

2.6.1 INCLUDEREA UNEI BIBLIOTECI ÎN IAR EMBEDDED WORKBENCH

Exemplu de integrare a unei biblioteci în **IAR Embedded Workbench**:

1. **Localizarea bibliotecii:** odată creată, biblioteca va fi disponibilă în directorul de build al proiectului, de obicei în **proiect\Debug\Obj\nume_biblioteca.r90.**
2. **Adăugarea bibliotecii în proiect:**
   - În **workspace**, faceți **click dreapta** pe **proiect** și selectați **Add → Add Files**;
   - Selectați **Library → Object Files**;
   - Navigați către locația bibliotecii (**nume_biblioteca.r90**) și adăugați fișierul ca bibliotecă.

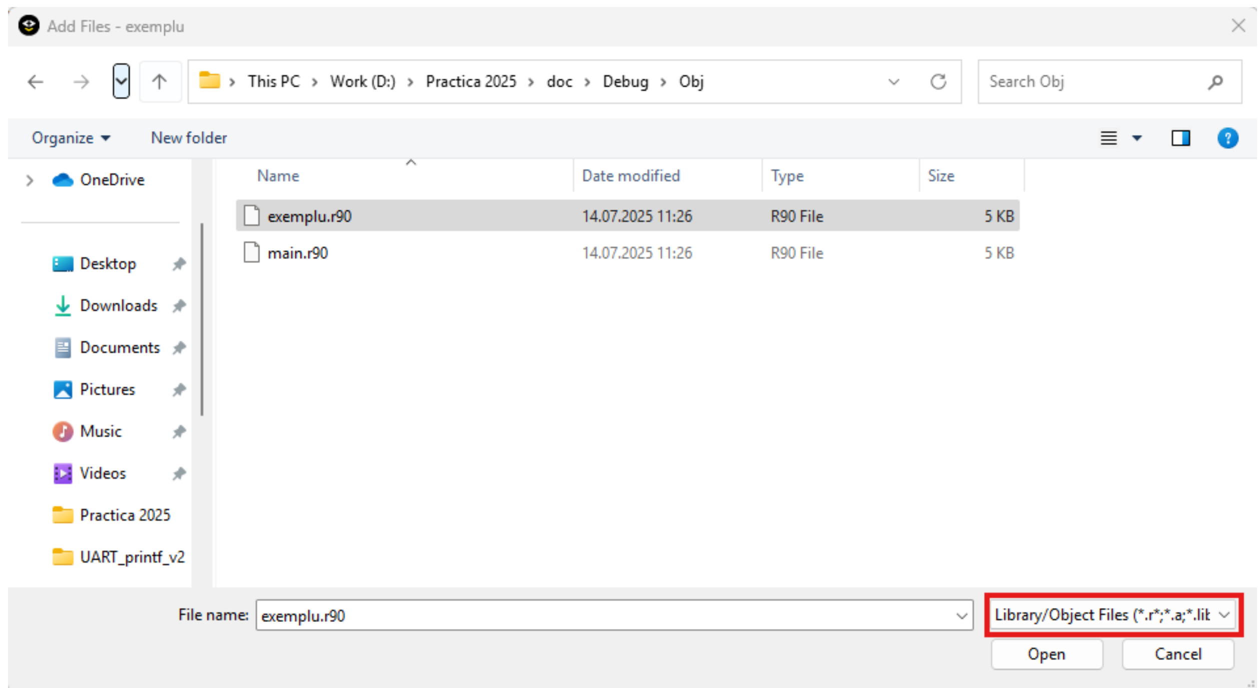


Figura 2.41 - Verificarea extensiei la adăugarea bibliotecii

3. **Compilarea proiectului (Make):** După ce biblioteca a fost adăugată, folosiți comanda **Make** pentru a **recompila** proiectul și a **verifica** integrarea corectă a bibliotecii.

2.7 BIBLIOGRAFIE

1. *"Embedded C", Wikipedia*
2. *"Embedded C++", Wikipedia*
3. *"IAR C/C++ Development Guide", IAR Systems*
4. *"IAR Embedded Workbench", IAR Systems*
5. *"MPLAB X IDE", Mikrochip Technology*

3. INTRODUCERE ÎN KIT-UL DE DEZVOLTARE

3.1 INTRODUCERE

Kit-ul de dezvoltare format din **SiBRAIN for ATmega1280** și **UNI Clicker de la Mikroe** este proiectat pentru a oferi o platformă versatilă și ușor de utilizat, dedicată dezvoltării rapide de aplicații embedded. Acest kit include un microcontroler **ATmega1280** pe placa **SiBRAIN**, care este completat de funcționalitățile multiple ale plăcii **UNI Clicker**. Acestea permit extinderea prin conectarea rapidă a diferitelor module periferice prin sloturi **mikroBUS**, asigurând astfel compatibilitate cu o gamă variată de senzori și dispozitive externe.

3.2 PREZENTAREA MICROCONTROLERULUI ATMEGA1280

ATmega1280 este un microcontroler **AVR** de **8 biți** cu arhitectură **RISC** avansată, care permite execuția majorității instrucțiilor în cicluri de ceas **unice**, oferind performanță ridicată. Printre caracteristicile sale principale se numără:

- frecvențe de **0-8 MHz** atunci când tensiunea de alimentare este între **2.7 V și 5.5 V;**
- frecvențe de **0-16 MHz** atunci când tensiunea de alimentare este între **4.5 V** și **5.5 V**;
- **32 KB** de memorie **Flash** și **2 KB** de **SRAM**;
- **1 KB** de **EEPROM** pentru stocarea datelor care trebuie păstrate după oprirea sistemului;
- **JTAG** pentru **debug** și **programare**;
- până la **53 de linii I/O programabile**, configurabile pentru a interacționa cu diverse periferice;
- interfețe integrate de comunicare serială: **USART**, **SPI**, **I²C** și altele, ideale pentru comunicarea cu module externe;
- **32 de registre** de uz general;
- **2 timere** de **8 biți** și **un timer** de **16 biți;**
- **4** canale de **PWM**;
- **ADC** cu **8 canale** de **10 biți**;
- **5 moduri** de **sleep**: **Idle**, **ADC Noise Reduction**, **Power-Save**, **Power-Down** și **Stand-By**.

3.3 PREZENTAREA PLĂCII SIBRAIN PENTRU ATMEGA1280

Placa **SiBRAIN for ATmega1280** oferă o platformă compactă, dar puternică, care facilitează dezvoltarea aplicațiilor embedded. Aceasta include conectori **dedicați** pentru acces la **toate interfețele** microcontrolerului, precum și pini de alimentare **stabilă** pentru funcționarea la **3.3 V** și **5 V**. Placa include, de asemenea, un **cristal extern** de **16 MHz** pentru precizie în generarea ceasului sistemului, precum și facilități pentru conectarea la interfețele **JTAG / SPI** pentru **debug** și **programare**.

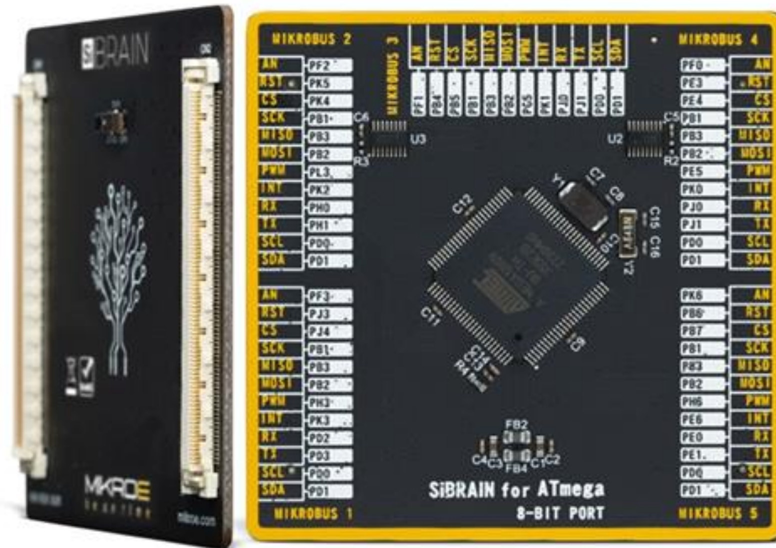


Figura 3.1 - SiBRAIN for ATmega 1280

3.4 PREZENTAREA PLĂCII UNI CLICKER

Placa **UNI Clicker** adaugă **funcționalități suplimentare** prin intermediul conectorilor **mikroBUS**. Aceasta permite conectarea rapidă a modulelor **click** care extind capabilitățile plăcii de bază. Printre funcțiile oferite de această placă se numără:

- **Managementul alimentării** pentru tensiuni de **3.3 V** și **5 V**, esențial pentru integrarea senzorilor și dispozitivelor externe. În plus, placa **UNI Clicker** include un **conector USB** utilizat pentru alimentare și programare, precum și un **conector de debug** care facilitează depanarea firmware-ului în timpul dezvoltării;
- Oferă acces la pini pentru interfețele de comunicare precum **UART**, **SPI**, **I²C** și **PWM**, prin intermediul sloturilor **mikroBUS**. Aceste interfețe sunt utilizate de modulele **click** conectate, care implementează efectiv funcționalitățile dorite.

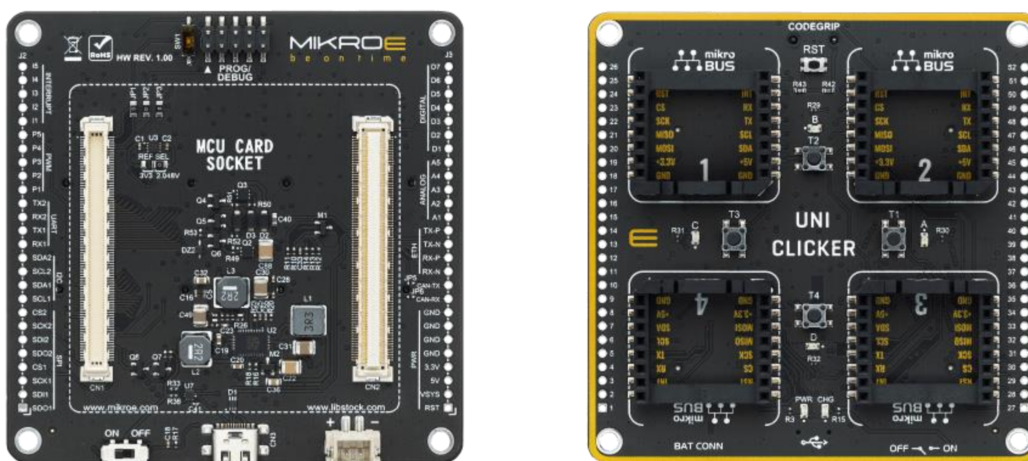


Figura 3.2 – UNI Clicker

3.5 NOȚIUNI DESPRE GPIO – PINI DE INTRARE / IEȘIRE GENERALĂ

3.5.1 DESCRIERE GENERALĂ

Microcontrolerul **ATmega1280** dispune de un număr mare de pini **GPIO** (**General Purpose Input / Output**) care pot fi configurați fie ca **intrări**, fie ca **ieșiri digitale**, în funcție de necesitățile aplicației. Acești pini sunt organizați în mai multe porturi (ex: **PORTA**, **PORTB**, **...**, **PORTL**), fiecare port având **8 biți** (pini numerotați de la **0** la **7**).

Exemple de utilizare:
- **Ieșire digitală**: controlul unui LED, a unui releu sau a unui tranzistor;
- **Intrare digitală**: citirea stării unui buton, a unui senzor digital sau a unui comutator.

Configurarea pinilor **GPIO** se face din firmware folosind registrele:
- **DDRn** (**Data Direction Register**): pentru stabilirea direcției (**intrare** sau **ieșire**);
- **PORTn**: pentru scrierea valorii logice (**HIGH / LOW**) pe pini când sunt configurați ca ieșire;
- **PINn**: pentru citirea stării logice de pe pini când sunt configurați ca intrare.

Acești pini sunt conectați fizic la exterior prin conectorii **CN1** și **CN2** de pe placa **SiBRAIN**, iar apoi, prin placa **UNI Clicker**, pot fi direcționați către sloturile **mikroBUS**, **LED-uri**, **butoane** sau **pini laterali**. Prin urmare, înțelegerea funcționării **GPIO** este **esențială** pentru **dezvoltarea aplicațiilor embedded** cu acest kit.

3.5.2 DESCRIERE PE SCURT A PINILOR ȘI UTILIZAREA LOR FRECVENTĂ

- **VCC – alimentarea digitală a microcontrolerului**

- **GND – masa = 0 V**

- **PortA (PA7...PA0) – Port I/O bidirecțional pe 8 biți**
  Servește și ca intrare analogică pentru **ADC** (**Analog to Digital Converter**). Poate fi configurat individual cu rezistențe de pull-up interne. Buffer-ele de ieșire au caracteristici de amplificare.

- **PortB (PB7...PB0) – Port I/O bidirecțional pe 8 biți cu rezistențe de pull-up interne**
  Poate îndeplini funcții speciale, inclusiv semnale pentru comunicație **SPI** sau semnale **PWM**.

- **PortC (PC7...PC0) – Port I/O bidirecțional pe 8 biți, cu funcții generale sau speciale.**
  Include linii pentru comunicare paralelă și alte funcții alternative.

- **PortD (PD7...PD0) – Port I/O bidirecțional pe 8 biți**
  Are funcții suplimentare ca semnale pentru temporizatoare sau intrări de întrerupere externă.

- **PortE (PE7...PE0) – Port I/O bidirecțional pe 8 biți**
  Include funcții speciale precum **UART** (**RXD**, **TXD**), semnale pentru temporizatoare și întreruperi externe.

- **PortF (PF7...PF0) – Port I/O bidirecțional pe 8 biți**
  Utilizat în principal ca intrare analogică pentru **ADC**. Merge și ca **I/O** general dacă **ADC-ul** nu este utilizat.

- **PortG (PG5...PG0) – Port I/O bidirecțional pe 6 biți**
  Are funcții speciale precum **controlul magistralei externe** sau **generare de semnale PWM**.

- **PortH (PH7...PH0) – Port I/O bidirecțional pe 8 biți**
  Utilizat și pentru funcții speciale precum semnale **PWM** sau **UART** suplimentar.

- **PortJ (PJ7...PJ0) și PortK (PK7...PK0) – Porturi I/O bidirecționale cu 8 biți**
  Utilizate ca **I/O** general sau pentru funcții de **comunicație** și **control**.

- **PortL (PL7...PL0) – Port I/O bidirecțional pe 8 biți**
  Utilizat pentru semnale de control pentru periferice.

- **RESET – Pin de resetare**
  O menținere a nivelului **0 logic** pentru un timp prestabilit declanșează **resetarea** microcontrolerului.

- **XTAL1**
  Intrare pentru oscilatorul extern sau pentru clock-ul intern.

- **XTAL2**
  Ieșire din oscilator.

- **AVCC – Alimentare pentru porturile analogice și pentru ADC**
  Trebuie conectat la **VCC**. Dacă **ADC** este utilizat, se recomandă conectarea printr-un filtru **trece-jos**.

- **AREF**
  Tensiune de referință pentru convertorul **analog-digital**.

- **GND și GND_ANALOG**
  Terminale de masă pentru partea **digitală** și partea **analogică**, respectiv.

- **PE0 / RXD0, PE1 / TXD0**
  Pini dedicați pentru comunicația **UART0**. **ATmega1280** dispune de **4 UART-uri** (**RXD0-3**, **TXD0-3**) disponibile pe diferite porturi.

3.5.3 PORTURILE DE INTRARE / IEȘIRE

Atunci când sunt folosite ca porturi generale digitale **I/O**, toate porturile au funcţionalitate de citire-modificare-scriere (“**True Read-Modify-Write**”). Aceasta înseamnă că, direcţia unui port poate fi schimbată fără schimbarea neintenţionată a direcţiei oricărui alt pin prin instrucţiunile **SBI** şi **CBI**. Toţi pinii porturilor au rezistenţă de pull-up individuală, selectabilă, cu rezerve de putere. Toţi pinii **I/O** au diodă de protecţie atât pentru **VCC** cât şi pentru masă, aşa cum se poate vedea în figura următoare:

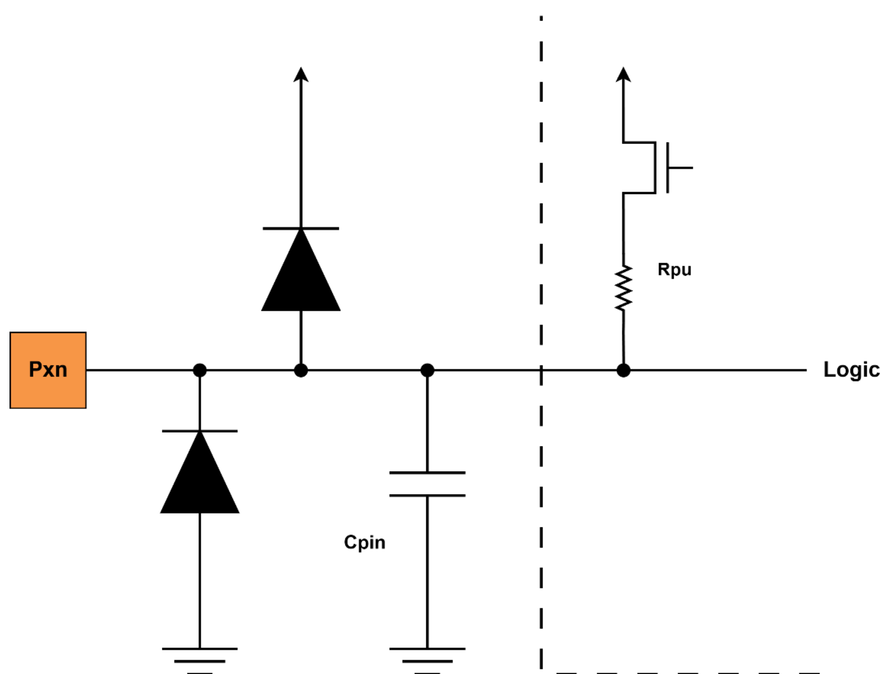


Figura 3.3 – Schema pinilor I/O

Porturile sunt de **I/O bidirecţionale** cu rezistenţe interne de pull-up **opţionale**.

3.5.4 CONFIGURAREA PINILOR

Pentru fiecare port de **I/O**, sunt alocate 3 locații de memorie, sub formă de registre pe **8 biți**:
- **PORTx (Data Register):** Registrul de date;
- **DDRx (Data Direction Register):** Registrul de direcție a datelor;
- **PINx (Port Input Pins):** Registrul de intrare al pinilor portului.

Registrul **PINx** poate fi doar citit, în timp ce **DDRx** și **PORTx** permit atât operații de citire, cât și de scriere. O diferență notabilă la modelele mai vechi era prezența bitului **PUD (Pull-up Disable)** în registrul **SFIOR**. La **ATmega1280**, acest bit a fost mutat în registrul **MCUCR (MCU Control Register)**, dar funcționalitatea sa de a dezactiva **global** rezistențele de pull-up pentru toate porturile rămâne aceeași.

Notația generică pentru acești pini este **DDxn**, **PORTxn** și **PINxn**, unde **"x"** reprezintă litera portului (de ex. A, B, C...), iar **"n"** este numărul bitului (0-7). Într-un program, trebuie folosită denumirea exactă, de exemplu **PORTB3** pentru **bitul 3** al **portului B**.

3.5.4.1 DDRX (DATA DIRECTION REGISTER)

Acest registru stabilește direcția fiecărui pin al portului, fie ca intrare, fie ca ieșire.
- Scrierea unui **0 logic** într-un bit din **DDRx** configurează pinul corespunzător ca **intrare**.
- Scrierea unui **1 logic** într-un bit din **DDRx** configurează pinul corespunzător ca **ieșire**.

Implicit, după un reset, toți pinii sunt configurați ca intrări (**DDRx = 0x00**).

Exemple:
- Pentru a seta **toți** pinii **portului A** ca intrări: **DDRA = 0x00**;
- Pentru a seta **toți** pinii **portului A** ca ieșiri: **DDRA = 0xFF**;
- Pentru a seta pinii **0**, **4**, **5** și **7** ai **portului B** ca ieșiri: **DDRB = 0xB1** (10110001 în binar).

3.5.4.2 PORTX (DATA REGISTER)

Registrul **PORTx** are un rol dublu, în funcție de direcția pinului setată în **DDRx**:
1. Când pinul este configurat ca **ieșire** (**DDxn = 1**):
   Valoarea scrisă în bitul corespunzător din **PORTxn** stabilește nivelul logic al pinului de ieșire.
   - Scrierea unui **1** în **PORTxn** va seta pinul de ieșire în stare logică **HIGH**.
   - Scrierea unui **0** în **PORTxn** va seta pinul de ieșire în stare logică **LOW**.
2. Când pinul este configurat ca **intrare** (**DDxn = 0**):
   Valoarea scrisă în **PORTxn** controlează rezistența de pull-up internă.
   - Scrierea unui **1** în **PORTxn activează** rezistența de pull-up internă pentru acel pin.
   - Scrierea unui **0** în **PORTxn dezactivează** rezistența de pull-up, iar pinul intră în starea de înaltă impedanță (**Hi-Z** sau **tri-state**).

Exemple:
- Setarea pinilor 0, 4, 5 și 7 ai portului B ca ieșiri și setarea lor la nivel logic HIGH:
  - setează pinii ca ieșiri.
    DDRB = 0xB1;
  - setează ieșirile la nivel logic 1
    PORTB = 0xB1;
- Setarea pinilor 0 și 1 ai portului A ca ieșiri și a pinului 2 ca intrare cu pull-up activat:
  - Pinii 0 și 1 ca ieșiri, restul intrări
    DDRA = 0x03;
  - Setează ieșirile la LOW și activează pull-up pentru pinul 2
    PORTA = 0x04;

3.5.4.3 PINX (PORT INPUT PINS)

Acest registru este folosit exclusiv pentru a citi **starea logică** a pinilor unui port, indiferent dacă sunt configurați ca **intrări** sau **ieșiri**. Registrul **PINx** este **read-only**. Citirea acestui registru **returnează** valoarea logică actuală de pe fiecare pin fizic.

Exemplu:

- Pentru a citi valorile pinilor portului C și a le stoca în variabila x:
  - Se setează toți pinii portului C ca intrări
    DDRC = 0x00;
  - Se copiază starea logică a pinilor portului C în variabila x
    x = PINC;

Tabelul de mai jos prezintă starea unui pin pentru diferite combinații de configurare ale registrelor. Bitul **PUD** se află în registrul **MCUCR** la ATmega1280.

DDxn	PORTxn	PUD (în MCUCR)	I/O	Pull-up	Descriere
0	0	X	Intrare	Nu	Tri-state (Hi-Z)
0	1	0	Intrare	Da	Pull-up activat
0	1	1	Intrare	Nu	Hi-Z (pull-up dezactivat global)
1	0	X	Ieșire	Nu	**LOW** (0 logic)
1	1	X	Ieșire	Nu	**HIGH** (1 logic)

Tabelul 3.1 - Starea unui pin pentru diferitele combinații ale regiștrilor

3.5.5 FUNCȚII INTRINSECI

Definiție

Funcțiile intrinseci oferă acces direct la instrucțiuni specifice ale procesorului, fiind extrem de utile pentru operațiuni critice din punct de vedere al timpului sau pentru optimizarea codului. Aceste funcții sunt de obicei compilate inline, adică sunt înlocuite direct cu secvența de instrucțiuni corespunzătoare, fără a implica un apel de funcție.

Funcție Intrinsecă	Descriere
`__delay_cycles(unsigned long)`	Inserează o întârziere precisă, dependentă de numărul de cicluri de ceas specificat.
`__disable_interrupt()`	Dezactivează întreruperile globale (generează instrucțiunea **CLI**).
`__enable_interrupt()`	Activează întreruperile globale (generează instrucțiunea **SEI**).
`__extended_load_program_memory(...)`	Returnează **un octet** din memoria de program. Este utilizată pentru a accesa adrese de memorie **mai mari** de **64 KB**.
`__load_program_memory(...)`	Returnează **un octet** din memoria de program. Se folosește pentru a accesa primii **64 KB** de memorie.
`__no_operation()`	Nu execută nicio operație (generează instrucțiunea **NOP**).
`__sleep()`	Pune microcontrolerul într-un mod de consum redus (generează instrucțiunea **SLEEP**).
`__swap_nibbles(unsigned char)`	Schimbă între ei cei 4 biți **inferiori** cu cei 4 biți **superiori** ai unui octet.
`__watchdog_reset()`	Resetează timer-ul watchdog pentru a preveni resetarea sistemului (generează instrucțiunea **WDR**).

Tabelul 3.2 - Funcțiile intrinseci

3.5.5.1 __DELAY_CYCLES()

```
void __delay_cycles(unsigned long cycles)
```

Această funcție generează o buclă de **întârziere** care consumă exact numărul de cicluri de ceas specificat ca parametru, fără a avea alte efecte secundare. Valoarea specificată trebuie să fie o constantă **cunoscută la momentul compilării**. Funcția este ideală pentru crearea de **întârzieri scurte** și **precise**. Deoarece procesorul este blocat într-o buclă de așteptare pe durata execuției, aceasta **nu este** o metodă **eficientă** pentru **întârzieri lungi** în aplicații complexe.

3.5.5.2 __DISABLE_INTERRUPT()

```
void __disable_interrupt(void)
```

Dezactivează global toate întreruperile mascabile prin generarea instrucțiunii de asamblare **CLI** (**Clear Global Interrupt Enable Bit**).

3.5.5.3 __ENABLE_INTERRUPT()

```
void __enable_interrupt(void)
```

Activează global întreruperile mascabile prin generarea instrucțiunii de asamblare **SEI** (**Set Global Interrupt Enable Bit**).

3.5.5.4 __EXTENDED_LOAD_PROGRAM_MEMORY() ȘI __LOAD_PROGRAM_MEMORY()

Aceste funcții sunt folosite pentru a **citi** date din memoria de program (**Flash**).

3.5.5.5 __NO_OPERATION()

```
void __no_operation(void)
```

Generează instrucțiunea **NOP** (**No Operation**). Această instrucțiune consumă un ciclu de ceas fără a modifica starea vreunui registru, fiind utilă pentru a crea **întârzieri foarte scurte** și **precise**, de un singur ciclu.

3.5.5.6 __SLEEP()

```
void __sleep(void)
```

Generează instrucțiunea **SLEEP**. Pentru a utiliza această funcție, trebuie mai întâi să **configurați** modul de **"sleep"** dorit în registrul **MCUCR** și apoi să **activați** bitul de **"sleep enable"**. Apelarea funcției va pune microcontrolerul în modul de **consum redus preselectat**, oprind procesorul până la apariția unei **întreruperi** sau a unui **eveniment de trezire**.

3.5.5.7 __SWAP_NIBBLES()

```
unsigned char __swap_nibbles(unsigned char value)
```

Această funcție **interschimbă** grupurile de biți **0-3** (**nibble-ul inferior**) cu biții **4-7** (**nibble-ul superior**) din octetul furnizat ca parametru și **returnează** noua valoare. Generează instrucțiunea **SWAP**. De exemplu, dacă **intrarea** este **0xA1** (**10100001**), **ieșirea** va fi **0x1A** (**00011010**).

3.5.5.8 __WATCHDOG_RESET()

```
void __watchdog_reset(void)
```

Generează instrucțiunea **WDR** (**Watchdog Reset**). Această funcție resetează timer-ul intern **Watchdog**. Trebuie apelată periodic pentru **a preveni resetarea** microcontrolerului de către Watchdog, în cazul în care acesta este activat.

3.6 COD EXEMPLU

O funcție de bază, comună în majoritatea aplicațiilor embedded, este **controlul LED-urilor**. În exemplul următor, vom implementa o secvență de **aprindere intermitentă** a unui **LED** (**LED blink**), pentru a ilustra modul în care pinii microcontrolerului **ATmega1280** sunt conectați la placa **UNI Clicker**.

3.6.1 URMĂRIREA UNUI PIN DE LA ATMEGA1280

3.6.1.1 CONEXIUNI PE PLACA SIBRAIN

Pe placa **SiBRAIN**, pinii **GPIO** ai microcontrolerului **ATmega1280** sunt conectați direct la conectorii externi **FX10A** (**CN1** și **CN2**), permițând astfel accesul la semnalele interne ale microcontrolerului. În schema electrică a plăcii **SiBRAIN for ATmega1280**, se poate observa clar această rutare a pinilor către exterior.

3.6.1.2 CONEXIUNI PE PLACA UNI CLICKER

Pe placa **UNI Clicker**, semnalul de la conectorul **FX10A** de pe **SiBRAIN** este preluat și dirijat prin circuite de rutare către sloturile **mikroBUS**, dar și către alte componente precum **LED-uri**, **butoane** sau **pini laterali accesibili**.

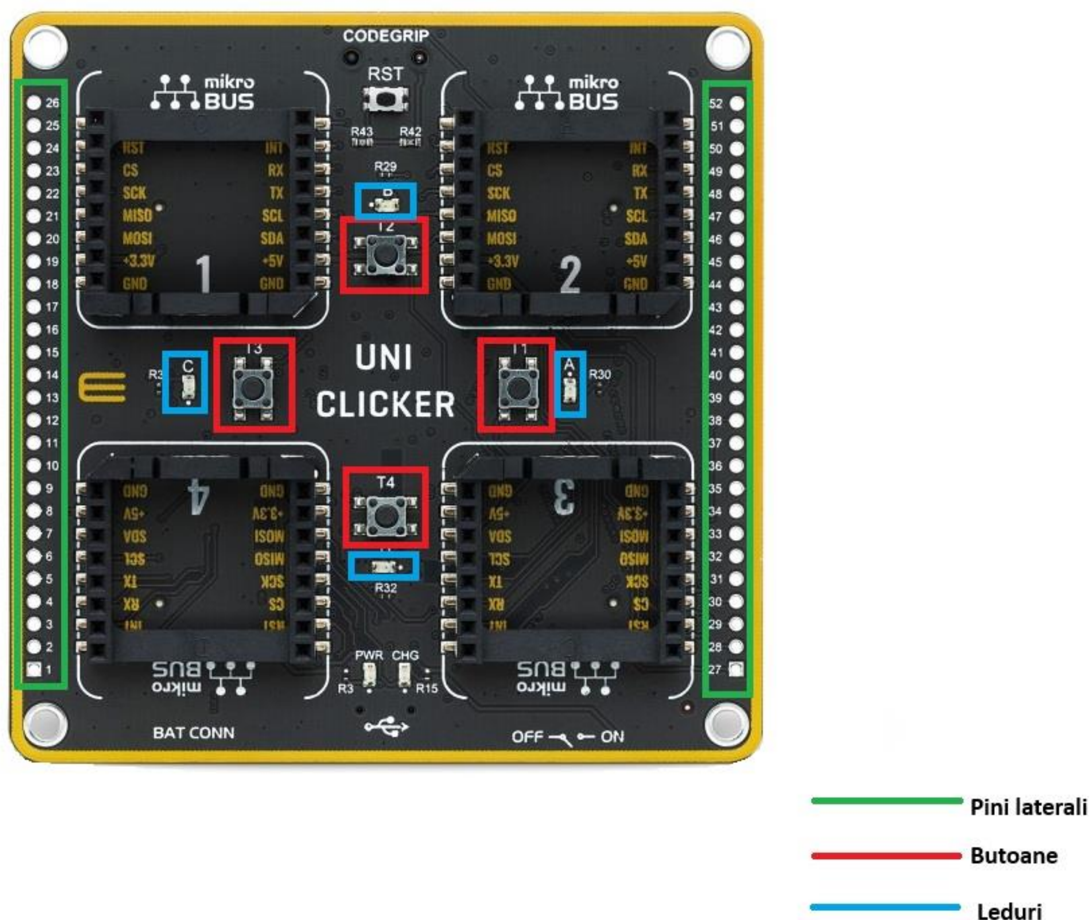


Figura 3.4 – Layout-ul plăcii UNI Clicker

3.6.2 EXEMPLU PRACTIC - URMĂRIREA PINULUI PA5

Să luăm următorul scenariu pentru a clarifica cum urmărim **conexiunile** de la **Atmega1280** la **mikroBUS**:

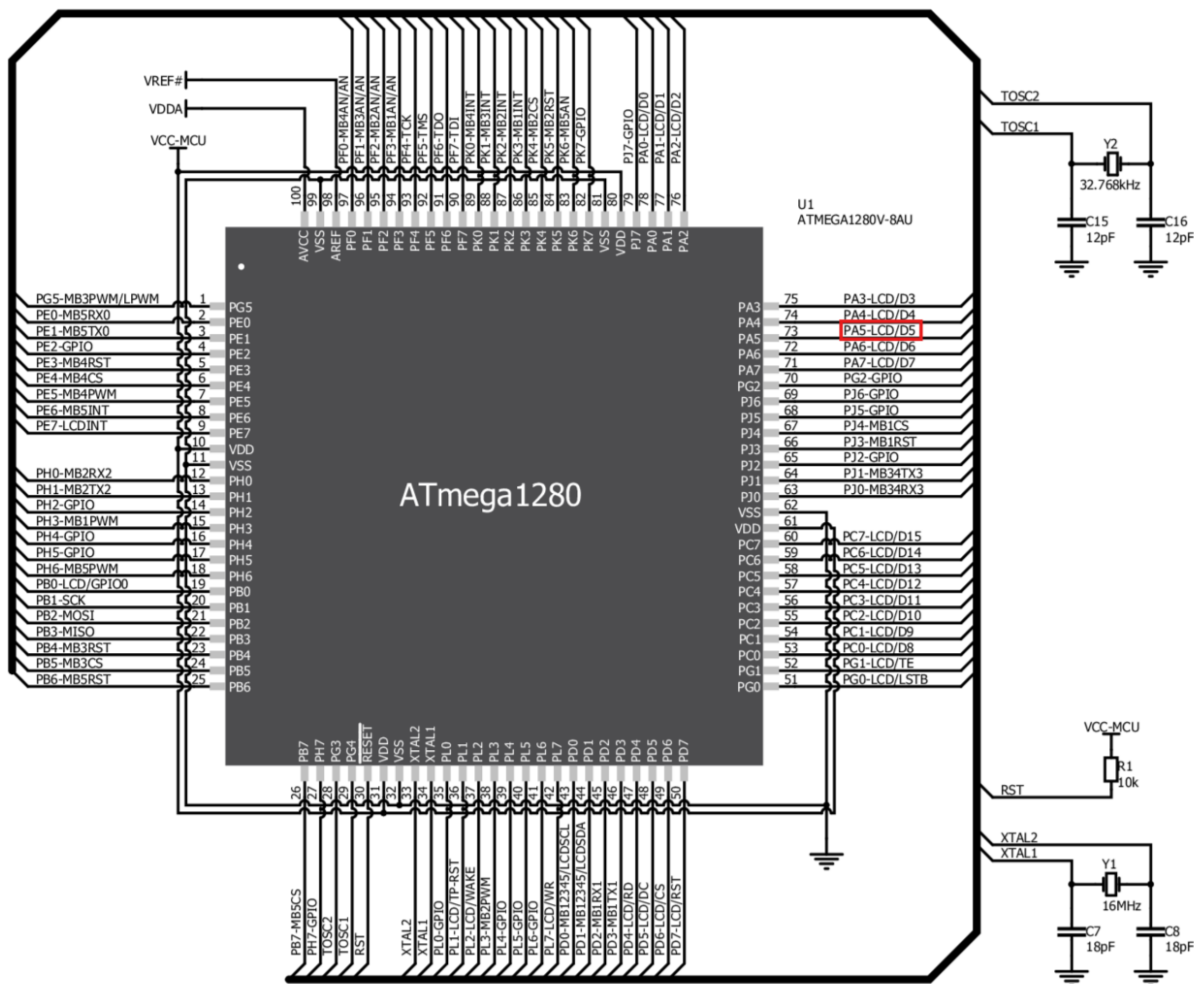


Figura 3.5 – Pinul PA5 al plăcii ATmega1280

Pe schema electrică a microcontrolerului **Atmega1280** din **datasheet**, pinul **PA5** este un pin **GPIO**, configurabil ca **ieșire** digitală.

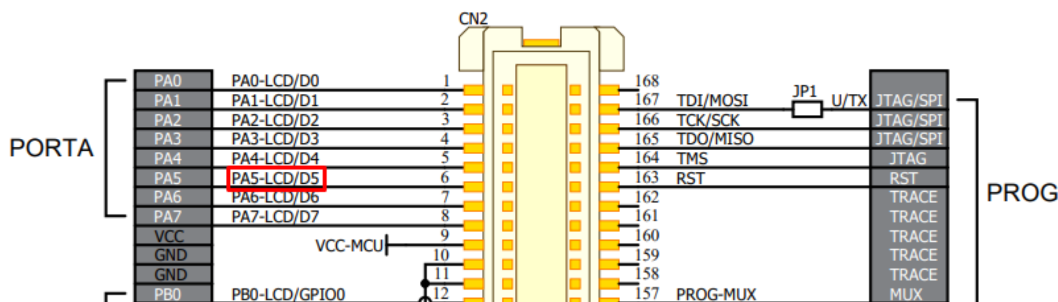


Figura 3.6 – Conexiunea pinului PA5 de pe SiBRAIN

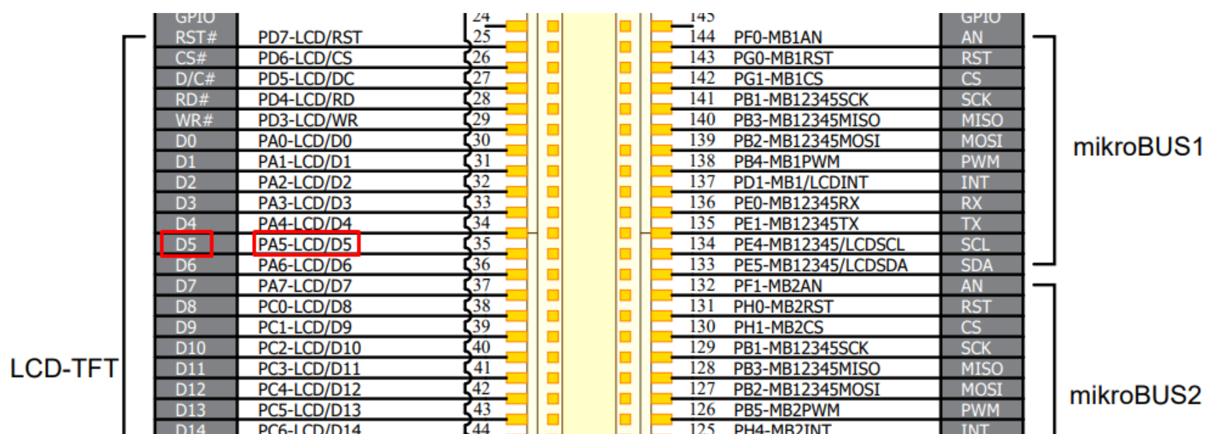


Figura 3.7 – Conexiunea pinului PA5 de pe SiBRAIN

Pe schema electrică a plăcii **SiBRAIN**, **PA5** este conectat **PORTA**, dar şi la **LCD-TFT** (**PA5 - LCD/D5**).

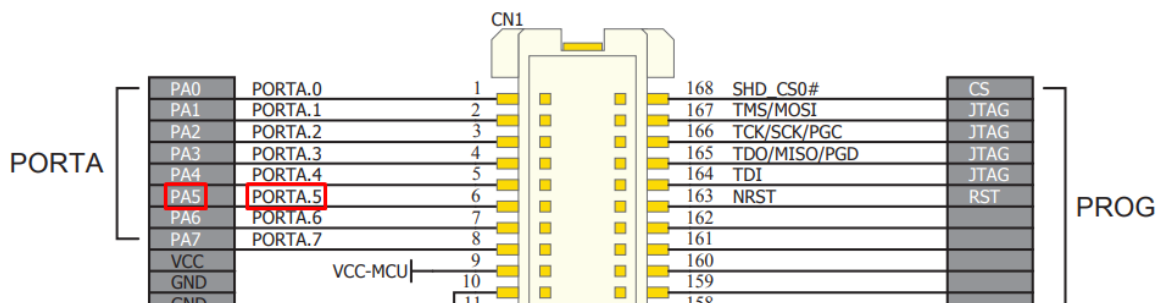


Figura 3.8 – Conexiunea pinului PA5 de pe UNI Clicker

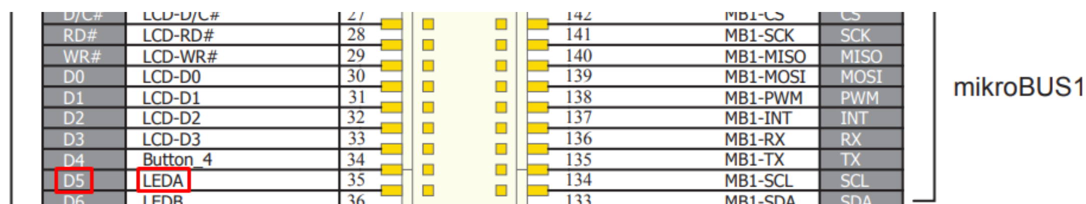


Figura 3.9 – Conexiunea pinului PA5 de pe UNI Clicker

Pe placa **UNI Clicker**, pinul respectiv se află pe conectorul **FX10A** (**CN1**) conectat la **PORTA5** (pinul 5 al portului A), dar şi la **LED-ul A** de pe **UNI Clicker**.

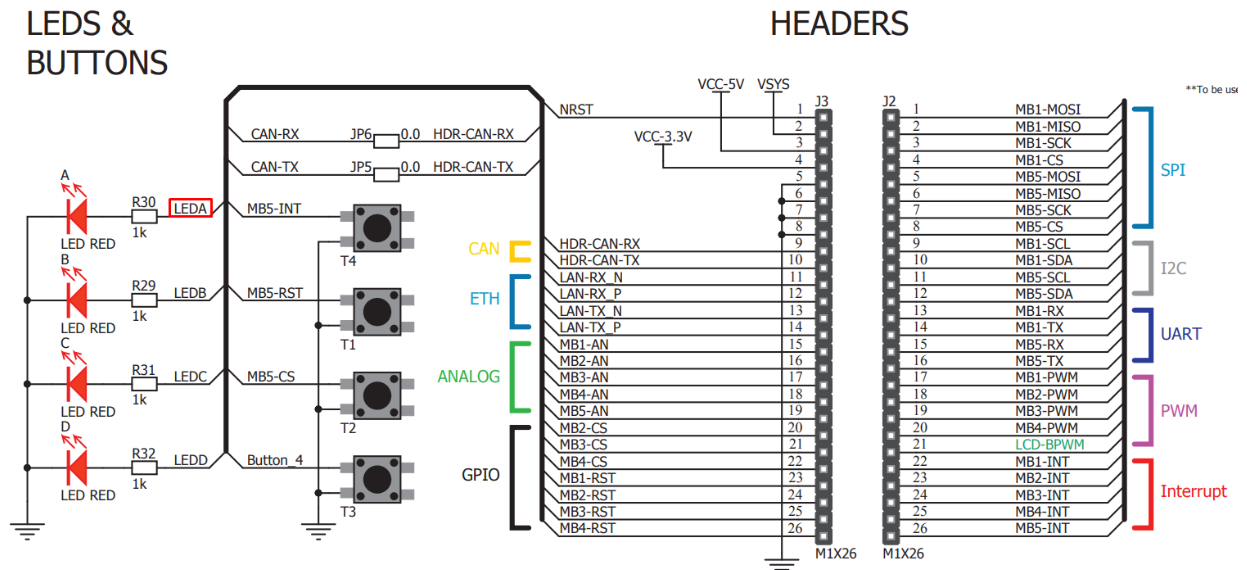


Figura 3.10 – Conexiunae D5-LED A de pe UNI Clicker

Prin analizarea schemelor electrice și a conexiunilor fizice dintre pinii **ATmega1280**, **SiBRAIN** și **UNI Clicker**, putem explica clar cum un semnal de la **microcontroler** ajunge la conectorii **mikroBUS** sau alte componente de pe **UNI Clicker**. Astfel, urmărirea unui pin este **esențială** pentru **înțelegerea** modului în care un semnal din firmware poate **controla** componente **externe** conectate la sloturile **mikroBUS**.

`exemplu_led.c`

Fișierul **`exemplu_led.c`** prezintă un mod simplu de a controla **LED-ul A**. În cod, se configurează pinul **PA5** al microcontrolerului **ATmega1280** ca **ieșire** și îl setează inițial pe nivel logic **0**. În bucla infinită **`while(1)`**, valoarea pinului este comutată (**toggle**), ceea ce face ca **LED-ul** conectat la **PA5** să **clipească**, cu o întârziere mică între schimbări.

```c
/*----------------------------------------------------------------------
 * Fișier: exemplu_led.c
 * Utilizat pentru exemplificarea controlării LED-ului A
 *--------------------------------------------------------------------*/

// Includes
#include <inavr.h>
#include <ioavr.h>

int main(void)
{
  // Se activează PA5 ca ieșire din ATmega1280
  DDRA |= (1 << PA5);
  // Se inițializează PA5 cu valoarea 0
  PORTA  &= ~(1 << PA5);

  while(1)
  {
      // Se comută valoarea pinului PA5
      PORTA ^= (1 << PA5);
      __delay_cycles(1000);
  }
}
```

3.6.3 URMĂRIREA UNUI SEMNAL DE INTRARE DE LA BUTONUL T1

În acest exemplu, urmărim traseul unui semnal de **intrare** de la butonul **T1** (**MB5-RST**) până la aprinderea **LED-ului A** de pe placa **UNI Clicker**, pentru a ilustra funcționarea unui pin **GPIO** configurat ca intrare digitală.

Pe placa **UNI Clicker**, pinul **MB5-RST** este conectat la butonul **T1**. La apăsarea butonului, semnalul este tras la masă (**GND**), iar microcontrolerul detectează această modificare de stare logică.

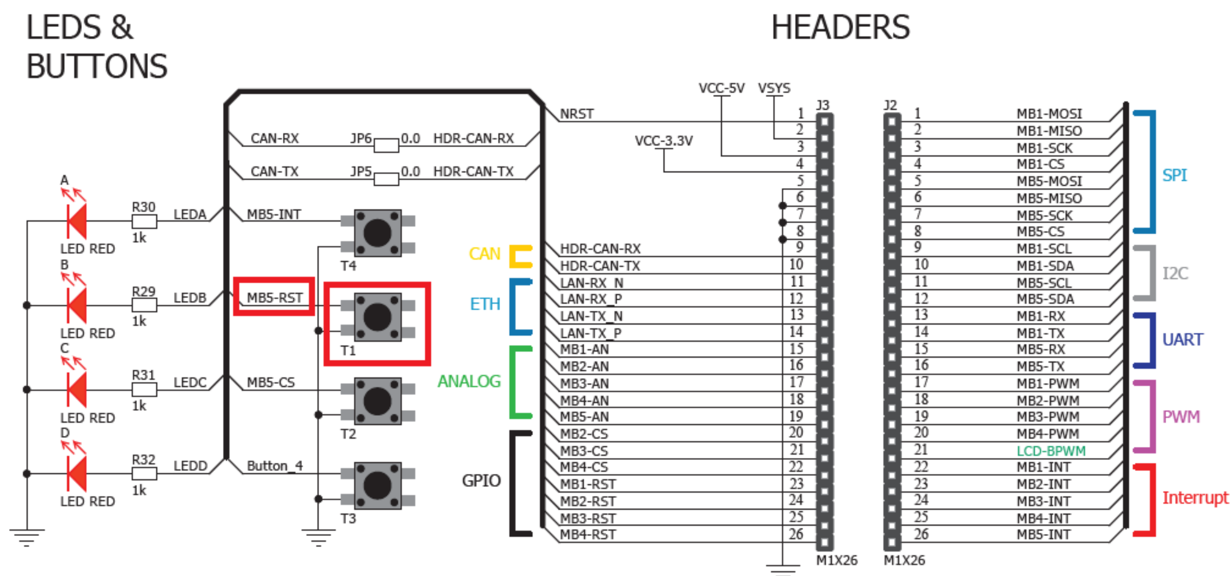


Figura 3.11 – Conexiunea MB5-RST (butonul T1) de pe UNI Clicker

Pinul **MB5-RST** este conectat la pinul **95**.

PC5	PORTC.5	74	95	MB5-RST	RST
PC6	PORTC.6	75	94	MB5-CS	CS
PC7	PORTC.7	76	93	MB5-SCK	SCK
PD0	PORTD.0	77	92	MB5-MISO	MISO
PD1	PORTD.1	78	91	MB5-MOSI	MOSI
PD2	PORTD.2	79	90	MB5-PWM	PWM
PD3	PORTD.3	80	89	MB5-INT	INT
PD4	PORTD.4	81	88	MB5-RX	RX
PD5	PORTD.5	82	87	MB5-TX	TX
PD6	PORTD.6	83	86	MB5-SCL	SCL
PD7	PORTD.7	84	85	MB5-SDA	SDA

Figura 3.12 – Conexiunea MB5-RST (butonul T1) de pe UNI Clicker

Pe schema electrică a plăcii SiBRAIN pinul 95 reprezintă pinul PB6:

PC4	PC4-LCD/D12	73	96	PK6-MB5AN	AN
PC5	PC5-LCD/D13	74	95	PB6-MB5RST	RST
PC6	PC6-LCD/D14	75	94	PB7-MB5CS	CS
PC7	PC7-LCD/D15	76	93	PB1-MB12345SCK	SCK
PD0	PD0-MB12345/LCDSCL	77	92	PB3-MB12345MISO	MISO
PD1	PD1-MB12345/LCDSDA	78	91	PB2-MB12345MOSI	MOSI
PD2	PD2-MB1RX1	79	90	PH6-MB5PWM	PWM
PD3	PD3-MB1TX1	80	89	PE6-MB5INT	INT
PD4	PD4-LCD/RD	81	88	PE0-MB5RX0	RX
PD5	PD5-LCD/DC	82	87	PE1-MB5TX0	TX
PD6	PD6-LCD/CS	83	86	PD0-MB12345/LCDSCL	SCL
PD7	PD7-LCD/RST	84	85	PD1-MB12345/LCDSDA	SDA

mikroBUS5

Figura 3.13 – Conexiunea MB5-RST de pe SiBRAIN

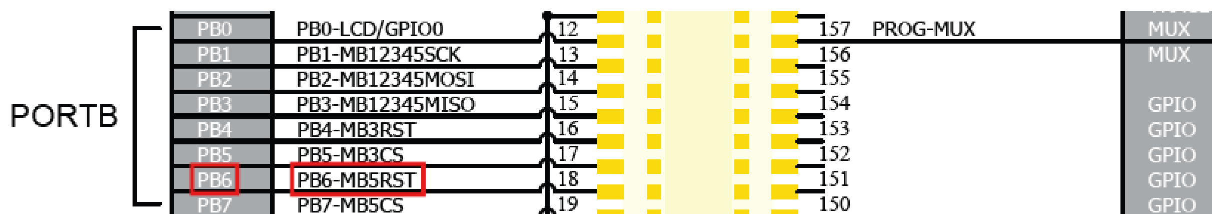


Figura 3.14 – Conexiunea MB5-RST de pe SiBRAIN

Iar aici vedem conexiunea **PB6** de pe microcontrolerul **ATmega1280**:

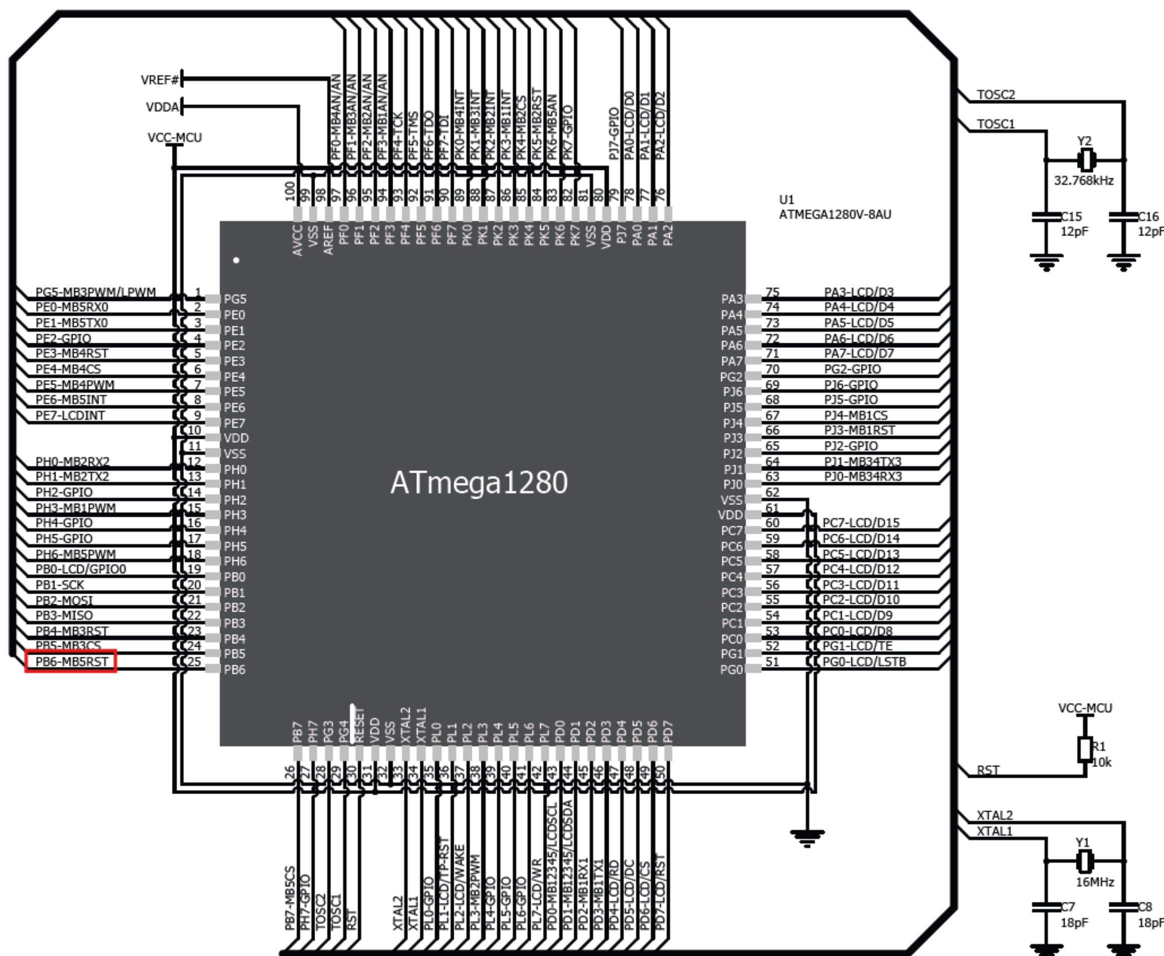


Figura 3.15 – Conexiunea PB6 de pe ATmega1280

Pe placa **SiBRAIN**, microcontrolerul **ATmega1280** are pinul **PB6** configurat ca pin **GPIO**. Acesta este conectat fizic la conectorii standard **FX10A** (**CN1** / **CN2**), care transmit semnalul mai departe către placa **UNI Clicker**.

Când butonul **T1** este apăsat, semnalul logic pe linia **PB6** devine **LOW**. Când butonul nu este apăsat, pinul este menținut pe **HIGH** prin activarea rezistenței interne de pull-up. Acest semnal este folosit de firmware pentru a comanda aprinderea unui **LED** (de exemplu, **LED-ul A** de pe **PA5**).

`exemplu_buton.c`

Fișierul **`exemplu_buton.c`** prezintă aprinderea **LED-ul PA5** la apăsarea butonului **T1**. Codul configurează pinul **PA5** ca ieșire (LED) și pinul **PB6** ca intrare (buton) cu rezistență internă de pull-up activată. În bucla infinită **`while(1)`**, verifică starea butonului: dacă acesta este apăsat (nivel **LOW**), **LED-ul** de pe **PA5** se aprinde; altfel, **LED-ul** rămâne stins.

```
/*-----------------------------------------------------------------------
 * Fișier: exemplu_buton.c
 * Utilizat pentru aprinderea LED-ului PA5 la apăsarea butonului T1
 *---------------------------------------------------------------------*/

// Includes
#include <inavr.h>
#include <ioavr.h>

int main(void)
{
  // Se activează PA5 ca ieșire din ATmega1280
  DDRA |= (1<<PA5);
  // PB6 ca intrare pentru a se citi butonul
  DDRB &= ~(1 << PB6);
  // Se activează rezistența internă de pull-up pe PB6
  PORTB |= (1 << PB6);

  while(1)
  {
      // Dacă butonul T1 este apăsat (LOW)
      if (!(PINB & (1 << PB6)))
      {
          // Se aprinde LED-ul
          PORTA |= (1 << PA5);
      }
      else
      {
          // Se stinge LED-ul
          PORTA &= ~(1 << PA5);
      }
  }
}
```

3.7 MODULE COMPATIBILE CU UNI CLICKER

3.7.1 BARGRAPH CLICK

BarGraph Click este o placă compactă ce include un **afișaj LED** sub formă de **bară grafică**, concepută pentru a oferi feedback vizual clar și uniform. Placa dispune de **10 LED-uri roșii** controlabile individual, inclusiv intensitatea luminoasă. Controlul se realizează prin intermediul a **2 registre de deplasare** de **8 biți**, **SN74HC595D**, produse de Texas Instruments.

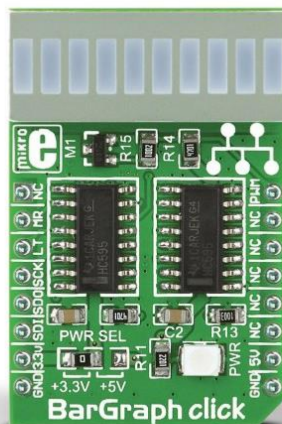


Figura 3.16 – BarGraph Click

Acest click este ideal pentru afișarea diverselor proprietăți de **semnal** sau **stare**, fiind perfectă pentru construirea de **VU-metere**, indicatori de **curent** / **tensiune**, poziția unui **encoder** sau orice altă proprietate ce poate fi reprezentată **grafic**. Afișajul cu **LED-uri** oferă o culoare uniformă și vizibilitate excelentă, făcând-o potrivită pentru o varietate de aplicații.

3.7.2 EEPROM CLICK

EEPROM Click este o placă de extensie compactă care oferă o soluție de **stocare de memorie nevolatilă**. Aceasta include componenta **FT24C08A**, o memorie **EEPROM** de **8192 biți** (**1KB**), ce utilizează o interfață serială **I²C**. Memoria este protejată împotriva scrierii hardware pentru întreaga capacitate, asigurând o siguranță sporită a datelor. Organizată intern în **1024 de cuvinte** a câte **8 biți** fiecare, această memorie este fabricată utilizând un proces **CMOS** avansat, care o face ideală pentru aplicații cu consum redus de energie și tensiune joasă.

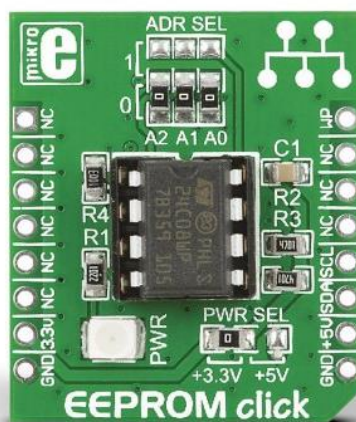


Figura 3.17 – EEPROM Click

FT24C08A suportă până la **un milion de cicluri complete** de citire / scriere / ștergere și oferă o retenție a datelor de peste **100 de ani**. **EEPROM Click** este soluția ideală pentru aplicații în care este necesară o stocare fiabilă de memorie nevolatilă.

3.7.3 RS232 ISOLATOR CLICK

RS232 Isolator Click este un **transceiver** dual complet izolat, proiectat pentru conversia sigură și ușoară de la **UART** la **RS232**, cu izolație galvanică. Semnalele digitale de intrare și ieșire sunt transmise peste bariera de izolare utilizând tehnologia **iCoupler** de la **Analog Devices**, unde înfășurările transformatorului de scară **IC** cuplează magnetic semnalele digitale, oferind izolația galvanică și rate de transfer de până la **460 Kbps**.

Figura 3.18 – RS232 Isolator Click

Acest modul este ideal pentru **izolarea galvanică** a semnalelor **RS232**, fiind util în medii dificile unde pot apărea descărcări electrostatice (**ESD**), cum ar fi atunci când **cablul RS232** este conectat și deconectat frecvent. În general, **RS232 Isolator Click** protejează circuitele logice sensibile ale controlerului **RS232** împotriva **șocurilor electrice** și **interferențelor** nedorite.

3.7.4 USB UART 4

USB UART 4 Click oferă o interfață de conversie de la **USB** la date seriale asincrone (**UART**), permițând dispozitivelor bazate pe microcontrolere să comunice ușor cu un computer personal. Este echipat cu **FT232RL**, un circuit integrat popular pentru conversia de la **USB** la **UART**, cunoscut pentru fiabilitatea și simplitatea sa, utilizat pe multe dispozitive **MikroElektronika**.

Figura 3.19 – USB UART 4 Click

Acest modul este ideal atunci când este necesară o conectare **simplă** și **eficientă** a liniilor **UART** la un computer. Poate fi folosit cu **orice terminal UART**, inclusiv cel integrat în compilatoarele **MikroElektronika**, facilitând **dezvoltarea** și **debugging-ul** aplicațiilor embedded.

3.7.5 7X10 B CLICK

7x10 B Click este o placă cu afișaj **LED dot matrix**, ideală pentru afișarea **cifrelor** sau a **literelor** într-un mod simplu și eficient. Această placă include **două module de afișaj LED dot matrix** cu elemente **LED** de tip punct rotund dispuse în matrice **7x5**. Afișajele produc modele clare și uniforme, deoarece elementele sunt **izolate optic**, prevenind scurgerile de lumină între celulele **LED** adiacente.

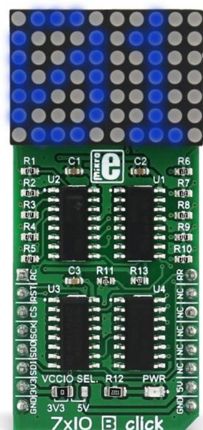


Figura 3.20 – 7x10 B Click

Timpul de pornire și oprire al celulelor din matrice este optimizat pentru o performanță fluidă și curată, fără efect de **pâlpâire** sau **întârziere**, asigurând o experiență de afișare de înaltă calitate.

3.7.6 LCD MONO CLICK

LCD Mono Click este o placă **click board** care utilizează afișajul **LS013B7DH03** de la **Sharp**, combinat cu microcontrolerul **EFM32** de la **Silicon Labs**, cunoscut pentru capacitățile sale de economisire a energiei. Această combinație permite dezvoltarea unei aplicații puternice de afișare, capabilă să controleze un afișaj de **128x128 pixeli**, consumând doar **2 µA** în timpul afișării unei imagini statice. Chiar și la actualizarea cadrului **o dată pe secundă**, consumul de curent poate fi **mai mic de 5 µA**.

Figura 3.21 – LCD Mono Click

LCD Mono Click este susținută de o bibliotecă compatibilă **mikroSDK**, care facilitează dezvoltarea software-ului. Această placă este complet testată și gata de utilizare într-un sistem echipat cu socket **mikroBUS**.

3.8 BIBLIOGRAFIE

1. *"8-bit AVR Microcontroller ATmega 1280 Datasheet", Microchip Technology*
2. *"Schematic for UNI Clicker", Mikroe*
3. *"Schematic for ATmega1280: SiBrain", Mikroe*
4. *"Legacy Products ", Mikroe*

4. ÎNTRERUPERI

4.1 UNDE SE FOLOSESC ÎNTRERUPERILE?

Întreruperile sunt folosite pe scară largă în sisteme embedded și electronice moderne pentru a răspunde rapid la evenimente externe sau interne, fără a irosi timp de procesare. Câteva exemple includ:

- **Tastaturi:** detectarea apăsării unei taste, fără a verifica constant starea lor în buclă;
- **Comunicare serială (I^2C, SPI, UART):** semnalizarea finalizării transmiterii / recepției unui byte;
- **Senzori:** notificarea microcontrolerului atunci când este disponibilă o nouă măsurătoare;
- **Sisteme de siguranță:** oprirea imediată a unui motor sau declanșarea unei alarme în caz de eroare;

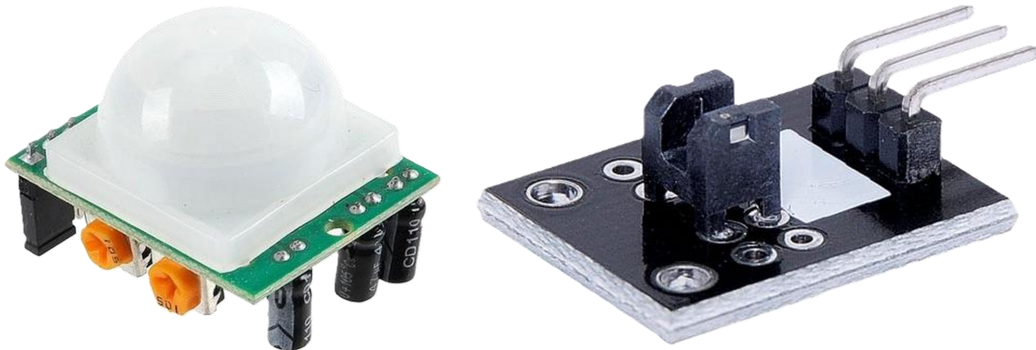


Figura 4.1 – Un senzor PIR de mișcare (stânga) și un senzor de lumină (dreapta), componente hardware care facilitează transmiterea unui semnal de întrerupere

4.2 CUNOȘTINȚE NECESARE

Pentru a putea parcurge acest capitol, este necesar să cunoașteți următoarele subiecte din capitolele anterioare, și nu numai:

- Noțiuni fundamentale de programare în C.
- Funcționarea microcontrolerelor.
- Configurarea și utilizarea perifericelor.
- Noțiuni de bază despre electronica digitală.

4.3 ABSTRACT

Acest capitol detaliază utilizarea întreruperilor (IRQ) în contextul microcontrolerului ATmega1280. Vor fi prezentate concepte de bază, tipurile de întreruperi disponibile, configurarea acestora și exemple de cod pentru gestionarea întreruperilor externe și interne.

4.4 HARDWARE ȘI SOFTWARE NECESAR

- ATMega1280 SiBRAIN;
- UNI Clicker;
- Atmel ICE;
- Generator de funcții programabil;
- IAR *Embedded Workbench* IDE 7.30.5;
- Osciloscop.

4.5 INTRODUCERE ÎN ÎNTRERUPERI

Definiție

O întrerupere reprezintă un semnal sincron sau asincron de la un periferic ce semnalizează apariția unui eveniment ce trebuie tratat de către procesor.

Tratarea de întreruperi are ca efect suspendarea firului normal de execuție al unui program și lansarea în execuție a unei rutine de tratare a întreruperii (RTI).

Întreruperile hardware au fost introduse pentru a se elimina buclele pe care un procesor ar trebui sa le facă în așteptarea unui eveniment de la un periferic. Folosind un sistem de întreruperi, perifericele pot atenționa procesorul în momentul producerii unei întreruperi (IRQ) acesta din urmă fiind liber să ruleze programul normal în restul timpului și să întrerupă execuția doar atunci când este necesar.

Înainte de a se lansa în execuție o RTI, procesorul trebuie să aibă la dispoziție un mecanism prin care să salveze starea în care se află în momentul apariției întreruperii. Aceasta se face prin salvarea într-o memorie, de cele mai multe ori organizată sub forma unei stive, a registrului contor de program (**Program Counter**), a registrelor de stare precum și a tuturor variabilelor din program care sunt afectate de execuția RTI. La sfârșitul execuției RTI, starea anterioară a registrelor este refăcută și programul principal este reluat din punctul de unde a fost întrerupt. Întreruperile sunt indispensabile în proiectarea unui sistem care să reacționeze corect și eficient în raport cu lumea exterioară. Faptul că au suport *hardware* oferă un timp de răspuns și *overhead* minimal.

În schimb, **întreruperile software** au un neajuns intrinsec. În primul rând, ele nu sunt portabile pe diferite procesoare și chiar pe diferite compilatoare. În al doilea rând, întreruperile pot determina numeroase erori software greu de identificat.

Tratarea întreruperilor implică un echilibru între timpii de răspuns și complexitatea codului. Dacă întreruperile sunt gestionate ineficient, ele pot conduce la stări de concurență problematică sau la întârzieri semnificative în execuția programului. De asemenea, ISR-urile trebuie să fie cât mai scurte și eficiente posibil pentru a minimiza timpul în care procesorul nu poate răspunde altor întreruperi. Un alt aspect important în proiectarea sistemelor cu întreruperi este managementul priorităților, asigurându-se că cele mai critice evenimente sunt gestionate cu prioritate maximă.

4.6 NOȚIUNI

Sistemele de întreruperi permit activarea sau dezactivarea acestora prin intermediul unor biți de mascare. Acest lucru este util pentru a preveni întreruperile în secțiunile critice ale codului.

Definiție

Latența reprezintă timpul care trece de la momentul în care apare o întrerupere și până când procesorul începe execuția RTI corespunzătoare.

Acesta poate fi influențată de mai mulți factori, inclusiv de arhitectura procesorului și de alte întreruperi în curs de procesare. Latența întreruperii nu este constantă și depinde de condițiile de execuție; pentru analiza sistemului se consideră, de obicei, **cea mai mare latență posibilă** (worst-case latency).

O întrerupere are două înțelesuri apropiate:

- Transferul hardware al controlului (saltul firului de execuție) către un vector de întrerupere pe baza înregistrării unui fenomen exterior procesului.
- Funcția de tratare a întreruperii, adică secvența de cod la care se ajunge plecând de la vectorul de întrerupere.

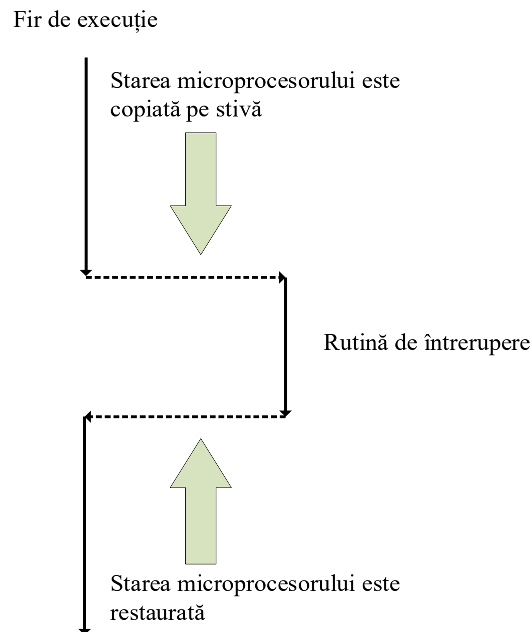


Figura 4.2 – Mecanismul de tratare al întreruperilor la nivel de microprocesor

Pentru a asocia o întrerupere cu o anumită rutină din program, procesorul folosește tabela vectorilor de întrerupere (TVI). Aceste adrese sunt predefinite şi sunt mapate în memoria din program într-un spaţiu contiguu (consecutiv, fără întreruperi) care alcătuieşte TVI. Adresele sunt setate în funcție de prioritatea lor, cu cât adresa este mai mică cu atât prioritatea este mai mare.

Nr. vector	Adresa programului (cuvinte)	Sursa	Definiția întreruperii
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	INT2	External Interrupt Request 2
5	0x0008	INT3	External Interrupt Request 3
6	0x000A	INT4	External Interrupt Request 4
7	0x000C	INT5	External Interrupt Request 5
8	0x000E	INT6	External Interrupt Request 6
9	0x0010	INT7	External Interrupt Request 7
10	0x0012	PCINT0	Pin Change Interrupt Request 0
11	0x0014	PCINT1	Pin Change Interrupt Request 1
12	0x0016	PCINT2	Pin Change Interrupt Request 2
13	0x0018	WDT	Watchdog Time-out Interrupt
14	0x001A	TIMER2 COMPA	Timer/Counter2 Compare Match A
15	0x001C	TIMER2 COMPB	Timer/Counter2 Compare Match B
16	0x001E	TIMER2 OVF	Timer/Counter2 Overflow
17	0x0020	TIMER1 CAPT	Timer/Counter1 Capture Event
18	0x0022	TIMER1 COMPA	Timer/Counter1 Compare Match A
19	0x0024	TIMER1 COMPB	Timer/Counter1 Compare Match B
20	0x0026	TIMER1 COMPC	Timer/Counter1 Compare Match C
21	0x0028	TIMER1 OVF	Timer/Counter1 Overflow
22	0x002A	TIMER0 COMPA	Timer/Counter0 Compare Match A
23	0x002C	TIMER0 COMPB	Timer/Counter0 Compare match B
24	0x002E	TIMER0 OVF	Timer/Counter0 Overflow
25	0x0030	SPI, STC	SPI Serial Transfer Complete

Tabelul 4.1 – Tabela vectorilor de întrerupere (TVI)

Se observă că **TVI** este mapat începând cu **adresa 0** a memoriei de program. Deși fiecare vector de întrerupere corespunde logic unei instrucțiuni (adică unui cuvânt de **2 octeți**), în practică, adresele sunt exprimate în **octeți**, nu în **cuvinte**. Astfel, fiecare **vector** ocupă **2 octeți** (**1 WORD**), iar adresele din tabel cresc din **2** în **2** (în octeți).

Prioritatea întreruperilor este determinată de ordinea acestora în tabel, astfel: cea **mai mare** prioritate este atribuită întreruperii de **RESET** (**adresa 0x0000**), urmând apoi întreruperea externă **INT0**, apoi restul întreruperilor în ordinea crescătoare a adreselor. Perifericele care pot genera întreruperi în cazul microcontrolerului **ATmega1280** includ: temporizatoare (**Timer0**, **Timer1**, etc.), interfața serială **USART**, convertorul analog-digital **ADC**, **EEPROM** (la finalizarea scrierii), comparatorul analogic, interfața serială **TWI**.

4.7 ACTIVAREA / DEZACTIVAREA ÎNTRERUPERILOR

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
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 4.3 – Registrul de stare (SREG)

O întrerupere este dezactivată daca s-a utilizat un suport hardware pentru a preveni declanșarea întreruperii. Suportul hardware este dat de 2 biți: unul specific fiecărui tip de întrerupere și unul ce se referă la toate întreruperile. Întreruperea de **Reset** face abatere de la această regulă prin faptul că nu poate fi prevenită. Saltul către un vector oarecare de întrerupere poate avea loc doar dacă cei doi biți au valoarea 1.

Pentru ATmega1280, bitul ce se referă la toate întreruperile se numește **I** și se află în registrul de stare al microprocesorului (bitul 7 din **Figura 4.3**). Pentru a-l modifica se pot utiliza următoarele instrucțiuni:

Mnemonică	Descriere	Operație	Număr de cicli	Funcții IAR
SEI	Global Interrupt Enable	I = 1	1	**_enable_interrupt()**
CLI	Global Interrupt Disable	I = 0	1	**_disable_interrupt()**

Tabelul 4.2 – Instrucțiunile de modificare ale bitului I

Instrucțiunile prezentate în tabel sunt utilizate pentru gestionarea întreruperilor la nivel global în cadrul arhitecturii AVR. Mnemonicile SEI (**Set Interrupt Enable**) și CLI (**Clear Interrupt Enable**) sunt comenzi în limbaj de asamblare care modifică bitul **I** (**Interrupt Enable**) din registrul de stare **SREG**, activând (I ← 1) sau dezactivând (I ← 0) procesarea întreruperilor. Aceste comenzi sunt executate într-un singur ciclu de ceas, fiind esențiale în controlul secvențelor critice de cod unde este necesară blocarea temporară a întreruperilor.

Funcțiile **`__enable_interrupt()`** și **`__disable_interrupt()`** oferă echivalentul în limbaj de programare C, specifice compilatoarelor precum IAR Embedded Workbench sau AVR-GCC, asigurând un mod portabil și lizibil de gestionare a întreruperilor globale în aplicații embedded. Activarea întreruperilor globale prin SEI nu are efect dacă întreruperile individuale nu sunt activate în registrele de control corespunzătoare (ex. EIMSK, PCICR). Astfel, utilizarea corectă a acestor instrucțiuni este esențială pentru sincronizarea precisă între perifericele controlate prin întreruperi.

Pentru activarea unei întreruperi externe pe microcontrolerul ATmega1280, este necesară configurarea corespunzătoare a registrelor de control implicate. În cazul întreruperii externe **INT0**, aceasta poate fi setată să răspundă la diverse condiții ale semnalului aplicat pe pinul **INT0**, inclusiv front crescător, front descrescător, nivel logic scăzut sau tranziție logică *(***toggle**). Modul de declanșare este determinat de valorile setate în biții **ISC01** și **ISC00** din registrul **EICRA**. Primul pas constă în configurarea modului de declanșare prin registrul **EICRA** (**External Interrupt Control Register A**), unde biții **ISC01** și **ISC00** determină tipul de tranziție ce va genera întreruperea. Pentru declanșare pe front crescător, ambii biți trebuie setați la **1**.

Apoi, întreruperea trebuie activată la nivel individual prin setarea bitului corespunzător din registrul EIMSK (**External Interrupt Mask Register**). Pentru **INT0**, acesta este bitul **0**. În cele din urmă, este necesară activarea întreruperilor la nivel global, folosind funcția **`__enable_interrupt()`**, care corespunde instrucțiunii SEI în limbajul de asamblare. Acest pas permite procesorului să răspundă la întreruperi.

`int0_avr.c`

Fragmentul de cod alăturat reprezintă un exemplu de configurare a unei întreruperi externe (INT0) pe AVR. Programul setează ca întreruperea să fie declanșată la front crescător al semnalului și activează întreruperile globale. Bucla infinită **`while(1)`** permite procesorului să aștepte evenimente externe.

```c
/*--------------------------------------------------------------------
 * Fișier: int0_avr.c
 * Utilizat pentru exemplificarea configurării întreruperii externe INT0
 *-------------------------------------------------------------------*/

#include <ioavr.h>
#include <inavr.h>
#include <intrinsics.h>

int main( void )
{
  // Configurarea întreruperii externe INT0 la frontul crescător
  EICRA |= (1<< ISC01) | (1 << ISC00);
  // Activarea întreruperii externe INT0
  EIMSK |= (1<< INT0);
  // Activarea întreruperilor globale
  __enable_interrupt();
  while(1)
  {
    /*
     * Buclă infinită pentru blocarea procesorului până la apariția unei
     * întreruperi
     */
  }
}
```

Se remarcă faptul că ultimul pas din procedura de inițializare este activarea întreruperilor globale, ceea ce respectă o ordine **bottom-up** în configurare: întâi se configurează evenimentul, apoi sursa, și la final se permite procesorului să răspundă global.

Fiecare tip de întrerupere este asociat cu un bit de stare care este setat de hardware atunci când apare evenimentul corespunzător. Pentru INT0, bitul respectiv este INTF0, aflat în registrul EIFR (**External Interrupt Flag Register**), pe poziția **0**. Resetarea acestui bit se face scriind valoarea **1** în poziția respectivă.

Notă: Contraintuitiv, resetarea bitului INTF0 se face scriind valoarea 1, nu 0.

Întreruperile au o prioritate determinată de poziția lor în tabela vectorilor de întrerupere. În cazul în care mai multe întreruperi sunt active simultan, cea cu **adresa de vector mai mică** va fi tratată prima. Funcțiile de tratare a întreruperilor nu pot fi întrerupte de altele cu prioritate mai mare în timpul execuției lor, decât dacă se activează manual acest comportament (prin nesting sau reentrancy). Se spune că o întrerupere este **reentrantă** dacă poate fi întreruptă de o alta cu prioritate superioară în timp ce ea este în execuție.

O rutină de întrerupere trebuie să se termine cu instrucțiunea RETI (**Return from Interrupt**), care restabilește starea anterioară a execuției și reia programul din punctul în care a fost întrerupt. Aceasta funcționează deoarece adresa curentă este salvată automat pe stivă de către microcontroler, înainte de saltul către vectorul de întrerupere.

4.8 REGISTRE PENTRU TRATAREA ÎNTRERUPERILOR EXTERNE

4.8.1 REGISTRUL EICRA

Bit	7	6	5	4	3	2	1	0	
(0x69)	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00	EICRA
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 4.4 – Registrul de control al întreruperilor externe A

Acest registru conține biții de control pentru configurarea sensibilității întreruperilor.

Biții **ISC01** și **ISC00** din registrul **EICRA** controlează sensibilitatea întreruperii externe **INT0**. Aceasta este activată atunci când bitul **I** din registrul **SREG** și masca corespunzătoare din **EIMSK** sunt setate. Nivelul logic sau tranzițiile de pe pinul **INT0** care declanșează întreruperea sunt definite conform tabelului de mai jos. Semnalul este eșantionat înainte de a fi evaluat, iar pentru modurile de tip front sau toggle, doar impulsurile care durează mai mult de o perioadă de ceas sunt sigure pentru a genera o întrerupere. Dacă se selectează modul „nivel logic 0", semnalul trebuie menținut pe **0** până la finalizarea execuției instrucțiunii curente pentru ca întreruperea să fie declanșată.

ISC01	ISC00	Descriere
0	0	Nivelul 0 logic pe INT0 generează o cerere de întrerupere.
0	1	Orice schimbare logică pe INT0 generează o cerere de întrerupere.
1	0	Pe frontul negativ al lui INT0 se generează o cerere de întrerupere.
1	1	Pe frontul pozitiv al lui INT0 se generează o cerere de întrerupere.

Tabelul 4.3 – Stările de declanșare ale întreruperii externe INT0

4.8.2 REGISTRUL EIMSK

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3D)	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	EIMSK
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 4.5 – Registrul de mascare a întreruperilor externe

Acest registru este răspunzător pentru activarea și dezactivarea întreruperilor externe.

- **Bitul 0 – INT0: Activare cerere de Întrerupere Externa 0**
  Când bitul **INT0** este setat (valoare logică 1) și bitul **I** din **SREG** este setat (valoare logică 1), întreruperea externă a pinului este activată. Biții de control al sensului întreruperii 0 / 1 (**ISC01** și **ISC00**) din registrul de control al întreruperilor externe A (**EICRA**) definesc dacă întreruperea externă este activată la frontul pozitiv și / sau negativ al pinului **INT0** sau la schimbare de nivel. Activarea pe pin va genera o cerere de întrerupere chiar dacă **INT0** este configurat ca pin de ieșire. Întreruperea corespunzătoare a cererii de întrerupere **0** este executate din vectorul de întrerupere **INT0**.

- **Bitul 4 – PCIE0: Bitul de activare întrerupere la schimbarea pinului 0**
  Când bitul **PCIE0** este setat (valoare logică 1) și bitul **I** din **SREG** este setat (valoare logică 1), întreruperea la schimbarea pinului **0** este activată. Orice schimbare pe oricare dintre pinii **PCINT7:0** activați va genera o întrerupere. Întreruperea corespunzătoare a cererii de întrerupere la schimbarea pinului este executată din vectorul de întrerupere **PCINT0**. Pinii **PCINT7:0** sunt activați individual prin registrul **PCMSK0**.

- **Bitul 5 – PCIE1: Activare întrerupere la schimbarea pinului 1**
  Când bitul **PCIE1** este setat (valoare logică 1) și bitul **I** din **SREG** este setat (valoare logică 1), întreruperea la schimbarea pinului **1** este activată. Orice schimbare pe oricare dintre pinii **PCINT15:8** activați va genera o întrerupere. Întreruperea corespunzătoare a cererii de întrerupere la schimbarea pinului este executată din vectorul de întrerupere **PCINT1**. Pinii **PCINT15:8** sunt activați individual prin registrul **PCMSK1**.

- **Bitul 6 – PCIE2: Activare întrerupere la schimbarea pinului 2**
  Când bitul **PCIE2** este setat (valoare logică 1) și bitul **I** din **SREG** este setat (valoare logică 1), întreruperea la schimbarea pinului **2** este activată. Orice schimbare pe oricare dintre pinii **PCINT23:16** activați va genera o întrerupere. Întreruperea corespunzătoare a cererii de întrerupere la schimbarea pinului este executată din vectorul de întrerupere **PCINT2**. Pinii **PCINT23:16** sunt activați individual prin registrul **PCMSK2**.

- **Bitul 7 – PCIE3: Activare întrerupere la schimbarea pinului 3**
  Când bitul **PCIE3** este setat (valoare logică 1) și bitul **I** din **SREG** este setat (valoare logică 1), întreruperea la schimbarea pinului **3** este activată. Orice schimbare pe oricare dintre pinii **PCINT30:24** activați va genera o întrerupere. Întreruperea corespunzătoare a cererii de întrerupere la schimbarea pinului este executată din vectorul de întrerupere **PCINT3**. Pinii **PCINT30:24** sunt activați individual prin registrul **PCMSK3**.

4.8.3 REGISTRUL EIFR

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	INTF7	INTF6	INTF5	INTF4	INTF3	INTF2	INTF1	INTF0	EIFR
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 4.6 – Registrul de indicatori ai întreruperilor externe

- **Bit 0 – INTF0: Indicator al Întreruperii Externe 0**
  Când o tranziție sau o schimbare logică pe pinul **INT0** declanșează o cerere de întrerupere, bitul **INTF0** se setează (**1**), microcontrolerul va sări la vectorul de întrerupere corespunzător. Indicatorul este resetat când rutina de întrerupere este executată. Alternativ, indicatorul poate fi resetat prin scrierea unei valorii de 1 logic în acesta. Acest indicator este întotdeauna resetat când **INT0** este configurat ca întrerupere pe nivel.

- **Bit 4 – PCIF0: Indicator al Întreruperii la Schimbarea Pinului 0**
  Când o schimbare logică pe oricare dintre pinii **PCINT7:0** declanșează o cerere de întrerupere, bitul **PCIF0** se setează (**1**). Dacă bitul **I** din **SREG** și bitul **PCIE0** din **EIMSK** sunt setați (**1**), microcontrolerul va sări la vectorul de întrerupere corespunzător. Indicatorul este resetat când rutina de întrerupere este executată. Alternativ, indicatorul poate fi resetat prin scrierea valorii de 1 logic în acesta.

- **Bit 5 – PCIF1: Indicator al Întreruperii la Schimbarea Pinului 1**
  Când o schimbare logică pe oricare dintre pinii **PCINT15:8** declanșează o cerere de întrerupere, bitul **PCIF1** se setează (**1**). Dacă bitul **I** din **SREG** și bitul **PCIE1** din **EIMSK** sunt setați (**1**), microcontrolerul va sări la vectorul de întrerupere corespunzător. Indicatorul este resetat când rutina de întrerupere este executată. Alternativ, indicatorul poate fi resetat prin scrierea valorii de 1 logic în acesta.

- **Bit 6 – PCIF2: Indicator al Întreruperii la Schimbarea Pinului 2**
  Când o schimbare logică pe oricare dintre pinii **PCINT23:16** declanșează o cerere de întrerupere, bitul **PCIF2** se setează (**1**). Dacă bitul **I** din **SREG** și bitul **PCIE2** din **EIMSK** sunt setați (**1**), microcontrolerul va sări la vectorul de întrerupere corespunzător. Indicatorul este resetat când rutina de întrerupere este executată. Alternativ, indicatorul poate fi resetat prin scrierea valorii de 1 logic în acesta. Acest bit este rezervat în ATmega1280 și va fi citit întotdeauna ca zero.

- **Bit 7 – PCIF3: Indicator al Întreruperii la Schimbarea Pinului 3**
  Când o schimbare logică pe oricare dintre pinii **PCINT30:24** declanșează o cerere de întrerupere, bitul **PCIF3** se setează (**1**). Dacă bitul **I** din **SREG** și bitul **PCIE3** din **EIMSK** sunt setați (**1**), microcontrolerul va sări la vectorul de întrerupere corespunzător. Indicatorul este resetat când rutina de întrerupere este executată. Alternativ, indicatorul poate fi resetat prin scrierea valorii de 1 logic în acesta.

4.8.4 REGISTRUL PCMSK0

Bit	7	6	5	4	3	2	1	0	
(0x6B)	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK0
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 4.7 – Registrul de mască pentru schimbarea pinului 0

- **Biții 7:0 – PCINT7:0: Mască de Activare a Întreruperii la Schimbarea Pinului 7:0**
  Fiecare bit **PCINT7:0** selectează dacă întreruperea la schimbarea pinului este activată pe pinul de **I/O** corespunzător. Dacă bitul **PCINT7:0** este setat și bitul **PCIE0** din **EIMSK** este setat, întreruperea la schimbarea pinului este activată pe pinul de **I/O** corespunzător. Dacă bitul **PCINT7:0** este șters, întreruperea la schimbarea pinului pe pinul de **I/O** corespunzător este dezactivată.

4.8.5 REGISTRUL PCMSK1

Bit	7	6	5	4	3	2	1	0	
(0x6C)	PCINT15	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	PCMSK1
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 4.8 – Registrul de mască pentru schimbarea pinului 1

- **Biții 7:0 – PCINT15:8: Mască de Activare a Întreruperii la Schimbarea Pinului 15:8**
  Fiecare bit **PCINT15:8** selectează dacă întreruperea la schimbarea pinului este activată pe pinul de **I/O** corespunzător. Dacă bitul **PCINT15..8** este setat și bitul **PCIE1** din **EIMSK** este setat, întreruperea la schimbarea pinului este activată pe pinul de **I/O** corespunzător. Dacă bitul **PCINT15:8** este șters, întreruperea la schimbarea pinului pe pinul de **I/O** corespunzător este dezactivată.

4.8.6 REGISTRUL PCMSK2

Bit	7	6	5	4	3	2	1	0	
(0x6D)	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	PCMSK2
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 4.9 – Registrul de mască pentru schimbarea pinului 2

- **Biții 7:0 – PCINT23:16: Mască de Activare a Întreruperii la Schimbarea Pinului 23..16**
  Fiecare bit **PCINT23:16** selectează dacă întreruperea la schimbarea pinului este activată pe pinul de **I/O** corespunzător. Dacă bitul **PCINT23:16** este setat și bitul **PCIE2** din **EIMSK** este setat, întreruperea la schimbarea pinului este activată pe pinul de **I/O** corespunzător. Dacă bitul **PCINT23:16** este șters, întreruperea la schimbarea pinului pe pinul de **I/O** corespunzător este dezactivată.

4.8.7 ANALIZA COMPARATIVĂ A ÎNTRERUPERILOR INTX VS PCINTX

În microcontrolerul ATmega1280, gestionarea evenimentelor externe se poate face prin două tipuri principale de întreruperi: întreruperile externe **INTx** și întreruperile la schimbare de pin **PCINTx**. Deși ambele permit detectarea modificărilor de semnal pe pini de intrare, ele diferă semnificativ ca funcționalitate, flexibilitate și mod de configurare.

Întreruperile **INTx** (de la **INT0** la **INT7**) sunt asociate cu pini dedicați și permit configurarea sensibilității la front ascendent, front descendent, schimbare logică sau nivel logic scăzut. Acestea oferă un control mai precis al detecției și sunt adesea utilizate în aplicații critice de timp real.

Pe de altă parte, întreruperile **PCINTx** (**Pin Change Interrupt**) sunt mai flexibile din punct de vedere al acoperirii, putând fi configurate pentru majoritatea pinilor de **I/O**, dar reacționează doar la orice schimbare logică (fie de la 0 la 1, fie de la 1 la 0), fără posibilitatea de a selecta tipul de tranziție. **PCINTx** sunt utile atunci când este necesară monitorizarea simultană a mai multor pini fără a consuma resurse suplimentare.

Prin urmare, alegerea între **INTx** și **PCINTx** depinde de cerințele aplicației, nivelul de precizie necesar în detecția evenimentelor și disponibilitatea pinilor corespunzători.

4.9 LATENȚĂ, JITTER ȘI TIMP DE RĂSPUNS LA ÎNTRERUPERI

4.9.1 LATENȚĂ

Definiție

Latența unei întreruperi reprezintă timpul scurs între momentul în care sursa de întrerupere solicită serviciu (prin activarea unei condiții de întrerupere) și momentul în care microcontrolerul începe execuția efectivă a rutinei de tratare (ISR – Interrupt Service Routine).

Latența este influențată de următorii factori:

- instrucțiunea curentă aflată în execuție (în special dacă este una multi-ciclu),
- activarea globală a întreruperilor (**SREG.I = 1**),
- prioritățile relative ale întreruperilor multiple aflate în așteptare.

Conform documentației oficiale (datasheet ATmega1280), **răspunsul minim la o întrerupere necesită 5 cicluri de ceas**, timp în care:

- program counter-ul este salvat pe stivă,
- execuția sare către adresa vectorului de întrerupere.

Această adresă vector indică de regulă un salt (**jmp**) către rutina de întrerupere, care mai consumă **încă 3 cicluri de ceas**, rezultând un total minim de **8 cicluri pentru intrarea efectivă în ISR**. Dacă întreruperea apare în timpul unei instrucțiuni multi-ciclu, aceasta este finalizată complet înainte de declanșarea ISR-ului.

În cazul în care microcontrolerul se află într-un mod de stand-by / sleep, timpul de răspuns este mărit cu încă **5 cicluri de ceas**, la care se adaugă și timpul specific de „wake-up” corespunzător modului de somn selectat.

La revenirea din întrerupere (care durează **5 cicluri de ceas**), procesorul:

- restaurează adresa salvată (3 octeți pentru PC),
- incrementează SP corespunzător,
- reactivează **flag-ul** global de întreruperi (setează bitul **I** din **SREG**).

Configurațiile software precum activarea / dezactivarea întreruperilor globale sau utilizarea modurilor de repaus pot introduce variații în latență, afectând astfel **jitter-ul** sistemului.

4.9.2 TIMP ESTIMAT

Pentru un sistem tactat la **16 MHz**, durata unui ciclu de ceas este de **62,5 ns**. Astfel, o latență minimă de **8 cicluri** înseamnă un timp de răspuns de aproximativ **500 ns**. În practică, din cauza variațiilor menționate anterior, se estimează o latență realistă între **2–3 μs**. Modul de declanșare se configurează prin biții **ISCxx** din registrul **EICRA**.

În funcție de modul de activare, întreruperile pot fi:
- **Edge-triggered** (pe front): reacționează la o tranziție (de exemplu, 0 → 1), fiind rapide și precise, dar sensibile la zgomot.
- **Level-triggered** (pe nivel): rămân active atât timp cât semnalul se menține, necesitând gestionarea atentă pentru a evita declanșări repetate neintenționate.

4.9.3 JITTER

Definiție

Jitter-ul definește variația temporală a momentului de răspuns la evenimente periodice sau întreruperi, în raport cu o referință temporală teoretic constantă.

Din perspectiva sistemelor embedded, jitter-ul afectează determinismul execuției și este considerat un parametru critic în aplicațiile real-time (RTOS sau bare-metal). Acest fenomen apare atunci când două întreruperi identice, generate în condiții aparent echivalente, sunt deservite la momente de timp diferite, rezultând în întârzieri variabile între impulsuri consecutive.

Principalele cauze ale jitter-ului includ:
- variația duratei de execuție a secvențelor de cod anterior întreruperii;
- blocarea temporară a întreruperilor (ex. în secțiuni critice sau în interiorul unei **ISR**);
- preempția de către întreruperi cu prioritate mai ridicată;
- latențe introduse de salvări/restaurări de context sau acces la magistrale partajate.

În aplicațiile deterministe (ex. control industrial, comunicații sincrone, aplicații medicale), jitter-ul trebuie:
- analizat prin măsurători precise (ex. osciloscop digital, analizor logic, timestamp-uri software);
- controlat prin optimizarea structurii codului (minimizarea ISR-urilor, prioritizare corectă);
- și minimizat prin configurarea adecvată a controllerului de întreruperi, izolarea sarcinilor critice și evitarea operațiilor blocante.

Un sistem embedded robust trebuie să asigure o variație minimă a timpului de răspuns la întreruperi, menținând astfel un comportament predictibil, repetabil și determinist, esențial în execuția aplicațiilor în timp real.

`interrupt_led_pulse.c`

Codul configurează microcontrolerul pentru a genera un semnal de referință și a aprinde un LED la fiecare depășire (overflow) a timer-ului 0. Timer-ul este setat în mod normal cu prescaler 64, iar întreruperea asociată (**`TIMER0_OVF_vect`**) declanșează simultan LED-ul și un puls pe pinul de referință. Întârzierea de aproximativ 10 μs simulează durata semnalului, după care LED-ul și semnalul de referință sunt stinse.

```c
/*----------------------------------------------------------------------
 * Fișier: interrupt_led_pulse.c
 * Utilizat pentru exemplificarea configurării întreruperii externe INT0
 *--------------------------------------------------------------------*/

#include <ioavr.h>
#include <intrinsics.h>
```

```c
/*----------------------------------------------------------------------
 * Public defines
 *--------------------------------------------------------------------*/

// Frecvența procesorului
#ifndef F_CPU
#define F_CPU 16000000UL
#endif

// Led ON / OFF / Init
#define LED_ON()     (PORTD |= (1 << 7))
#define LED_OFF()    (PORTD &= ~(1 << 7))
#define LED_INIT()   (DDRD |= (1 << 7))

// Puls pe semnal ON / OFF / Init
#define REF_ON()     (PORTC |= (1 << 0))
#define REF_OFF()    (PORTC &= ~(1 << 0))
#define REF_INIT()   (DDRC |= (1 << 0))

// ISR pentru Timer0 Overflow
#pragma vector = TIMER0_OVF_vect

/*
 * Funcția generează un puls de durată fixă la fiecare overflow al
 * Timer0, folosit pentru debugging sau sincronizare.
 */
__interrupt void Timer0_Overflow_ISR(void) {
    REF_ON();     // Puls pe semnalul de referință (pentru osciloscop)
    LED_ON();     // Aprindere LED

    __delay_cycles(10000);  // Întârziere ~10 µs (la 16 MHz)

    LED_OFF();    // Stingere LED
    REF_OFF();    // Final semnal trigger
}

void main(void) {
    LED_INIT();   // Configurare pin LED
    REF_INIT();   // Configurare pin referința

    TCCR0A = 0x00;

    // Timer0: normal mode, prescaler 64
    TCCR0B = (1 << CS01) | (1 << CS00);
    TIMSK0 = (1 << TOIE0);      // Activare overflow interrupt
    __enable_interrupt();       // Activare întreruperi globale

    while (1) {
     /*
      * Buclă infinită pentru blocarea procesorului până la apariția unei
      * întreruperi
      */
    }
}
```

4.10 TIPURI DE ÎNTRERUPERI: HARDWARE ȘI SOFTWARE

4.10.1 GENERAREA UNEI ÎNTRERUPERI HARDWARE

În această secțiune este ilustrată utilizarea unei întreruperi **hardware** generate de perifericul Timer / Counter0 al microcontrolerului **ATmega1280**. Modulul **Timer0** este configurat în mod de operare **Normal Mode** (incrementare simplă de la 0 la 255), utilizând un prescaler de 1024, ceea ce determină o frecvență de **overflow** scăzută, adecvată pentru aplicații demonstrative.

La fiecare depășire a valorii maxime de 8 biți (0xFF), se generează o întrerupere hardware, declanșând astfel executarea automată a unei rutine de serviciu a întreruperii (**ISR**) asociată vectorului **TIMER0_OVF_vect**. Această întrerupere este gestionată intern de unitatea de gestionare a întreruperilor (**Interrupt Controller**), fără intervenția explicită a codului principal.

În cadrul ISR-ului, se implementează un mecanism de contorizare software, care permite executarea unei acțiuni doar după un număr definit de **overflow-uri** (în acest caz, 30). Această tehnică este utilă pentru reducerea frecvenței acțiunilor declanșate, păstrând totodată precizia temporizării oferită de timer-ul hardware.

Concret, la fiecare 30 de întreruperi **TIMER0_OVF**, se inversează complet starea logică a portului **PORTD**, oferind astfel o ieșire observabilă (ex. LED-uri conectate la pinii portului) și ilustrând funcționarea asincronă a întreruperilor față de bucla principală (**main loop**), care rămâne goală.

Această abordare evidențiază avantajele întreruperilor hardware:
- eliminarea necesității de polling constant;
- separarea clară între logica de control a timpului și logica principală;
- reacție deterministă la evenimente hardware;
- eficiență în consumul de resurse CPU.

`led_blink.c`

Codul configurează microcontrolerul pentru a folosi **Timer0** cu întreruperi de overflow. La fiecare depășire a pragului setat (**`COUNTER_THRESHOLD`**), rutina de întrerupere inversează starea pinilor de pe **PORTD**, ceea ce duce la aprinderea / stingerea alternativă a LED-urilor conectate. Practic, se generează un semnal periodic pe **PORTD**, controlat de **Timer0** și contorul software.

```
/*------------------------------------------------------------------------
 * Fișier: led_blink.c
 * Utilizat pentru generarea unui semnal periodic
 *----------------------------------------------------------------------*/

// Includes
#include <ioavr.h>
#include <intrinsics.h>

/*-------------------------------------------------------------------------
 * Public defines
 *-----------------------------------------------------------------------*/

// Pragul de întreruperi pentru Timer0
#define COUNTER_THRESHOLD 30


// ISR pentru Timer0 Overflow
#pragma vector = TIMER0_OVF_vect

/*
 * Funcția generează un semnal mai lent (vizibil) derivat din întreruperile
 * rapide ale Timer0.
 */
```

```
__interrupt void Timer0_Overflow_ISR(void) {
    // Static reține valoarea între întreruperi
    static unsigned int interrupt_counter = 0;
    // Se incrementează contorul la fiecare întrerupere
    interrupt_counter++;
    /*
     * Se definește pragul la care vrem sa se întâmple acțiunea, cu cât
     * numărul este mai mare cu atât lipirea e mai lentă
     */
    if (interrupt_counter >= COUNTER_THRESHOLD) {
        // Se inversează starea pinilor de pe PORTD
        PORTD = ~PORTD;
        // Se resetează contorul pentru a începe o noua numărătoare
        interrupt_counter = 0;
    }
}

void main(void) {
    // Se configurează Port D ca ieșire
    DDRD = 0xFF;
    PORTD = 0x00;
    /*
     * Se configurează Timer0 pentru a genera o întrerupere la overflow
     * Prescaler = 1024
     */
    TCCR0B = (1 << CS02) | (1 << CS00);
    // Se activează întrerupere de overflow pentru Timer0
    TIMSK0 = (1 << TOIE0);


    // Activarea globală a întreruperilor
    __enable_interrupt();

    while (1) {
    /*
     * Buclă infinită pentru blocarea procesorului până la apariția unei
     * întreruperi
     */
    }
}
```

4.10.2 GENERAREA UNEI ÎNTRERUPERI SOFTWARE

Atât timp cât fenomenul cauză al întreruperii este de natură externă întreruperea este de tip **hardware**. Există și posibilitatea ca printr-o metodă **software** să se satisfacă condiția de întrerupere. O întrerupere declanșată pe această cale va fi de tip **software**. De exemplu, dacă o întrerupere externă este activată (**INT0, INT1, INT2**) și pinul corespunzător este setat ca ieșire, atunci întreruperea poate fi declanșată prin scrierea pinului.

int0_ext_interrupt.c

Codul configurează microcontrolerul pentru a folosi întreruperea externă **INT0** pe pinul **PD0**, declanșată la frontul pozitiv al semnalului de intrare. La fiecare întrerupere, rutina asociată incrementează o variabilă de test, iar în bucla principală se face toggling pe **PORTD** pentru a vizualiza schimbarea stării pinilor (de ex. LED-uri conectate). Astfel, se demonstrează modul de configurare și utilizare a întreruperilor externe și contorizarea evenimentelor generate de acestea.

```
/*-----------------------------------------------------------------------
 * Fișier: int0_ext_interrupt.c
 * Utilizat pentru configurarea întreruperilor externe
 *---------------------------------------------------------------------*/

// Includes
#include <ioavr.h>
#include <intrinsics.h>

/*
 * Funcția configurează întreruperea externă INT0 pe frontul pozitiv.
 * Aceasta setează toți pinii PORTD ca ieșire și aprinde inițial toate
 * LED-urile, după care activează întreruperile globale.
 */
void init_interrupt_INT0(void) {
    // Se setează toți pinii PORTD ca ieșire (LED-uri, test)
    DDRD = 0xFF;
    // Inițial, toate LED-urile sunt aprinse (logica inversă)
    PORTD = 0xFF;

    // Se configurează INT0 pe front pozitiv
    EICRA |= (1 << ISC01) | (1 << ISC00);
    // Se activează întreruperea INT0
    EIMSK |= (1 << INT0);
    // Se activează întreruperile globale
    __enable_interrupt();
}

int main(void)
{
    // Se configurează PORTD ca ieșire și inițializare LED-uri
    DDRD = 0xFF;
    PORTD = 0xFF;
    // Se configurează INT0 pe front pozitiv și se activează întreruperea
    EICRA |= (1 << ISC01) | (1 << ISC00);
    EIMSK |= (1 << INT0);
    __enable_interrupt();

    // Buclă infinită pentru generarea unei întreruperi software
    while (1)
    {
      PORTD = ~PORTD;
    }
}

// ISR pentru INT0
#pragma vector=INT0_vect

/*
 * Rutina de întrerupere asociată cu INT0. La declanșarea întreruperii,
 * variabila statică 'test' este incrementată. Această variabilă poate fi
 * utilizată pentru test sau debug.
 */
```

```
__interrupt void INT0_ISR(void) {
    static unsigned char test = 0;
    test += 1;
}
```

4.10.3 ANALIZA RUTINEI DE ÎNTRERUPERE LA NIVEL DE COD DE ASAMBLARE

Codul dezasamblat al rutinei de întrerupere **INT0_ISR** evidențiază pașii efectuați de compilator pentru a asigura tratamentul corect al unei întreruperi externe pe vectorul **INT0**. La intrarea în **ISR**, sunt salvați pe stivă registrii de uz general (**R16, R17, R18**) utilizați în cadrul funcției, precum și registrele speciale **SREG** (Status Register) și **RAMPZ** (utilizat pentru accesarea memoriei extinse).

Operația efectivă constă în încărcarea valorii unei variabile statice (`test`), incrementarea acesteia cu o unitate și scrierea valorii actualizate în memorie. La finalul rutinei, se restaurează contextul anterior prin retragerea registrelor de pe stivă și se revine din întrerupere cu instrucțiunea **`RETI`**, care reface automat starea de execuție a programului principal.

`int0_isr.asm`

Această secvență ilustrează respectarea convențiilor standard de salvare / restaurare a contextului într-o întrerupere, demonstrând o execuție predictibilă și izolată a codului asociat evenimentului hardware.

```
INT0_ISR:
    ST      -Y, R18         ; Salvează R18 pe stivă
    ST      -Y, R17         ; Salvează R17 pe stivă
    ST      -Y, R16         ; Salvează R16 pe stivă

    IN      R17, 0x3F       ; Salvează registrul SREG în R17
    IN      R18, 0x3B       ; Salvează RAMPZ în R18
    LDS     R16, ??test     ; Prima instrucțiune scrisă de programator
    INC     R16             ; Incrementare cu 1
    STS     ??test, R16     ; Ultima instrucțiune scrisă de programator

    OUT     0x3B, R18       ; Restaurează RAMPZ
    OUT     0x3F, R17       ; Restaurează SREG

    LD      R16, Y+         ; Restaurează R16 din stivă
    LD      R17, Y+         ; Restaurează R17 din stivă
    LD      R18, Y+         ; Restaurează R18 din stivă

    RETI                    ; Returnare din întrerupere
```

4.10.4 ANALIZA ÎNTRERUPERILOR DE TIP NESTING

Acest program demonstrează funcționarea întreruperilor **nesting** pe microcontrolerul ATmega1280, folosind întreruperi externe declanșate **software**:

- **INT1** este asociată unei sarcini cu **prioritate mică** – aprinderea LED-ului A (PA5) timp de 2 secunde.
- **INT0** este asociată unei sarcini cu **prioritate mare** – aprinderea LED-ului B (PA6) timp de 1 secundă.

În bucla principală (**`main`**), întreruperile sunt declanșate periodic (la fiecare 5 secunde) prin manipularea pinilor **PD1** (pentru **INT1**) și **PD0** (pentru **INT0**). În cadrul întreruperii **INT1**, se activează nesting-ul cu **`__enable_interrupt()`**, permițând astfel ca **INT0** să o întrerupă dacă se declanșează în timpul execuției.

Comportamentul de nesting este ușor observabil:

- **LED-ul A** se aprinde timp de 2 secunde (**INT1**).
- Dacă în timpul acesta este activată **INT0**, **LED-ul B** se aprinde timp de 1 secundă și întrerupe execuția **LED-ului A**.
- După terminarea execuției lui **INT0**, **LED-ul A** continuă să rămână aprins până la finalul celor 2 secunde.

Întârzierile sunt realizate cu `__delay_cycles()` pentru precizie, fără utilizarea de temporizatoare hardware. Acest cod evidențiază clar conceptul de întreruperi prioritare și **nesting**, atât vizual, cât și prin instrumente de măsurare (ex. osciloscop).

`nesting.c`

Codul configurează microcontrolerul pentru a demonstra întreruperi externe cu priorități diferite și **nesting**. **INT0** (prioritate mare) și **INT1** (prioritate mică) aprind alternativ LED-urile conectate pe **PORTA**. **ISR-ul** cu prioritate **mare** poate întrerupe **ISR-ul** cu prioritate **mică**, demonstrând comportamentul de întreruperi imbricate (**nesting**) și controlul vizual al priorităților prin LED-uri.

```
/*----------------------------------------------------------------------
 * Fișier: nesting.c
 * Utilizat pentru configurarea întreruperilor imbricate (nesting)
 *--------------------------------------------------------------------*/

// Includes
#include <ioavr.h>
#include <intrinsics.h>

/*----------------------------------------------------------------------
 * Public defines
 *--------------------------------------------------------------------*/

// Pinii corespunzători LED-urilor
#define LED_A_PIN   5 // Controlat de INT1 (prioritate mică)
#define LED_B_PIN   6 // Controlat de INT0 (prioritate mare)

// LED A ON / OFF / Init / Toggle
#define LEDS_INIT()       (DDRA |= (1 << LED_A_PIN) | (1 << LED_B_PIN))
#define LED_A_ON()        (PORTA |= (1 << LED_A_PIN))
#define LED_A_OFF()       (PORTA &= ~(1 << LED_A_PIN))
#define LED_A_TOGGLE()    (PORTA ^= (1 << LED_A_PIN))

// LED B ON / OFF / Toggle
#define LED_B_ON()        (PORTA |= (1 << LED_B_PIN))
#define LED_B_OFF()       (PORTA &= ~(1 << LED_B_PIN))
#define LED_B_TOGGLE()    (PORTA ^= (1 << LED_B_PIN))

// Sarcina cu prioritate mare
#pragma vector = INT0_vect

/*
 * Funcția aprinde LED-ul A pentru ~2 secunde. Având prioritate mică,
 * permite nesting și poate fi întrerupt de ISR-ul cu prioritate mare.
 */
__interrupt void HighPriority_ISR(void) {
    LED_B_ON();
    __delay_cycles(16000000); // Delay de o secundă
    LED_B_OFF();
}

// Sarcina cu prioritate mică
#pragma vector = INT1_vect

/*
 * Funcția aprinde LED-ul B pentru ~1 secundă. Având prioritate mare,
 * aceasta poate întrerupe ISR-ul cu prioritate mică.
 */
__interrupt void LowPriority_ISR(void) {
    __enable_interrupt(); // Se permite întreruperilor să fie imbricate
    LED_A_ON();
```

```
    __delay_cycles(24000000); // Delay de 2 secunde
    LED_A_OFF();
}
void main(void) {
    LEDS_INIT();

    // Configurare Întreruperi INT0 si INT1 pentru declanșare software
    DDRD  |= (1 << PD0) | (1 << PD1);
    PORTD &= ~((1 << PD0) | (1 << PD1));
    // Se configurează INT0 pe front pozitiv și se activează întreruperea
    EICRA |= (1 << ISC01) | (1 << ISC00) | (1 << ISC11) | (1 << ISC10);
    EIMSK |= (1 << INT0) | (1 << INT1);
    // Se activează întreruperile globale
    __enable_interrupt();

    while (1) {
        __delay_cycles(5 * 1600000); // 5 secunde
        // INT1 (prioritate mică)
        PORTD |= (1 << PD1);
        PORTD &= ~(1 << PD1);
        //INT0 (prioritate mare)
        PORTD |= (1 << PD0);
        PORTD &= ~(1 << PD0);
    }
}
```

4.10.5 IMPLEMENTAREA UNUI ALGORITM DE DEBOUNCE SOFTWARE

debounce.c

Acest cod implementează un algoritm de **debounce software** pentru a filtra zgomotul electric de la o intrare mecanică (**buton T4 / PE6**), asigurând o singură tranziție logică per apăsare. Metoda se bazează pe **polling temporizat** în bucla principală, utilizând o bază de timp precisă generată de **Timer0**. Acesta este configurat în modul CTC pentru a declanșa o întrerupere la fiecare **1 ms**, incrementând un contor global (**system_millis**). La detectarea unei tranziții pe pinul de intrare, programul înregistrează momentul și intră într-o perioadă de așteptare de **50 ms**. O acțiune (comutarea **LED-ului A**) este validată și executată doar dacă starea pinului rămâne stabilă pe toată durata acestui interval, eliminând astfel citirile false cauzate de contactele mecanice oscilante. **LED-ul B** oferă feedback vizual, fiind activ doar pe durata intervalului de **debounce**.

```
/*------------------------------------------------------------------------
 * Fișier: debounce.c
 * Utilizat pentru citirea unui buton cu debounce software
 *----------------------------------------------------------------------*/

// Includes
#include <ioavr.h>
#include <intrinsics.h>
#include <stdint.h>

/*------------------------------------------------------------------------
 * Public defines
 *----------------------------------------------------------------------*/

// Buton T4
#define BUTTON_PIN_REG  PINE
#define BUTTON_DDR      DDRE
#define BUTTON_PORT     PORTE
#define BUTTON_PIN      6
```

```c
// LED A pe PA5
#define LED_A_DDR       DDRA
#define LED_A_PORT      PORTA
#define LED_A_PIN       5

// LED B pe PA6
#define LED_B_DDR       DDRA
#define LED_B_PORT      PORTA
#define LED_B_PIN       6

/*-----------------------------------------------------------------------
 * Global variables
 *---------------------------------------------------------------------*/

// Variabila pentru stocarea timpului scurs, similar cu millis()
volatile unsigned long system_millis = 0;

// Variabile pentru starea debounce

// Starea anterioară a butonului (1 = neapăsat)
uint8_t last_button_state = 1;

// Starea stabilă a butonului
uint8_t debounced_button_state = 1;

// Timpul de la ultimul debounce
unsigned long last_debounce_time = 0;
// Timpul de așteptare debounce (în ms)
const unsigned int debounce_delay = 50;

// ISR pentru TIMER0 (ceasul sistemului)
#pragma vector = TIMER0_COMPA_vect

__interrupt void Millis_ISR(void) {
    system_millis++;
}

// Funcția de inițializare timer
void init_millis_timer(void) {
    /*
     * Configurare Timer0 mod CTC pentru a genera o întrerupere la 1 ms
     * F_CPU = 16 MHz. Prescaler = 64. F_timer = 16 MHz / 64 = 250 kHz.
     * Pentru 1 ms (1 kHz), OCR0A = (250 kHz / 1 kHz) - 1 = 249.
     */
    TCCR0A |= (1 << WGM01); // Mod CTC
    TCCR0B |= (1 << CS01) | (1 << CS00); // Prescaler 64
    OCR0A = 249;
    TIMSK0 |= (1 << OCIE0A); // Activează întreruperea pe Compare Match A
}

/*
 * Funcția configurează butonul (PE6) ca intrare cu pull-up și LED-urile
 * (PA5, PA6) ca ieșiri, pornind cu ambele stinse.
 */
void init_gpio(void) {
    // Configurare Buton (PE6) ca intrare cu pull-up
    BUTTON_DDR &= ~(1 << BUTTON_PIN);
    BUTTON_PORT |= (1 << BUTTON_PIN);

    // Configurare LED-uri (PA5, PA6) ca ieșiri
    LED_A_DDR |= (1 << LED_A_PIN);
```

```
    LED_B_DDR |= (1 << LED_B_PIN);

    // Se opresc LED-urile inițial
    LED_A_PORT &= ~(1 << LED_A_PIN);
    LED_B_PORT &= ~(1 << LED_B_PIN);
}

int main(void) {
    init_gpio();
    init_millis_timer();
    __enable_interrupt();

    while (1) {
        /*
         * Citirea stării actuale a butonului
         * Butonul conectat la GND este "0" (apăsat) și "1" (neapăsat),
         * datorită pull-up-ului.
         * Se inversează logica pentru a avea 1 = apăsat, 0 = neapăsat.
         */
        uint8_t reading = !(BUTTON_PIN_REG & (1 << BUTTON_PIN));

        // Se detectează o schimbare de stare (începutul zgomotului)
        if (reading != last_button_state) {

        // Dacă starea s-a schimbat, se resetează cronometrul de debounce
            last_debounce_time = system_millis;
            /*
             * Se aprinde LED-ul B pentru a semnaliza că s-a intrat în
             * perioada de debounce
             */
            LED_B_PORT |= (1 << LED_B_PIN);
        }

        // Se verifică dacă a trecut suficient timp pentru stabilizare
        if ((system_millis - last_debounce_time) > debounce_delay) {
        // LED-ul B e stins, fiindcă perioada de debounce s-a încheiat
            LED_B_PORT &= ~(1 << LED_B_PIN);

            /*
             * Se verifică dacă starea stabilă este diferită de starea
             * anterioara stabilă
             */
            if (reading != debounced_button_state) {
                debounced_button_state = reading;

                // Se verifică dacă noua stare stabilă este "apăsat"
                if (debounced_button_state == 1) {
                    // Se comută starea LED-ului A
                    LED_A_PORT ^= (1 << LED_A_PIN);
                }
            }
        }
        // Se actualizează starea anterioară pentru următoarea iterație
        last_button_state = reading;
    }
}
```

4.10.6 SCHIMBAREA TABELEI DE VECTORI DE ÎNTRERUPERE

Sistemul implementează o mașină de stări cu două moduri de operare distincte. Comutarea între aceste moduri se realizează prin manipularea directă a bitului **IVSEL** (**Interrupt Vector Select**) din **MCUCR**, conform secvenței specificate în datasheet-ul dispozitivului, care necesită setarea prealabilă a bitului **IVCE** (**Interrupt Vector Change Enable**).

- **Modul Aplicație (IVSEL = 0): IVT** este mapată la adresa de start a memoriei Flash (`0x0000`). O întrerupere externă, **INT6**, este configurată pe front descrescător și la activare invocă o **ISR** definită static prin directiva de compilare **`#pragma vector`**. Această rutină execută o secvență de control **I/O** specifică aplicației.
- **Modul Bootloader (IVSEL = 1): IVT** este relocată la adresa de start a secțiunii de boot, predefinită prin fuse bits **BOOTSZ** la `0x1E000`. Pentru a popula această a doua tabelă, s-a utilizat un fișier sursă în limbaj de asamblare (**`.s90`**) care, prin directiva ASEGN, plasează la adresa absolută **`0x1E000`** o structură de salturi (**JMP**). În acest mod, vectorul **INT6** va redirecționa execuția către o altă **ISR**, specifică bootloader-ului, chiar dacă sursa fizică a întreruperii rămâne aceeași.

Tranziția între moduri este gestionată de o a doua întrerupere, **PCINT0**. Vectorul acesteia a fost implementat în ambele tabele de întreruperi pentru a garanta funcționalitatea de comutare indiferent de starea curentă a bitului **IVSEL**. Astfel, proiectul validează un mecanism robust și eficient pentru izolarea și execuția contextuală a codului critic în sisteme embedded.

`bootloader_app_switch.c`

Codul configurează microcontrolerul pentru a lucra în două moduri: **aplicație** și **bootloader**. Trecerea între cele două se face printr-un buton dedicat, care generează o întrerupere și schimbă tabela de vectori prin bitul **IVSEL**, în timp ce **LED-ul C** indică modul curent. În modul **aplicație**, apăsarea butonului de acțiune activează o rutină care pulsează **LED-ul A**. În modul **bootloader**, același buton declanșează o rutină ce pulsează **LED-ul B**. Astfel, funcționarea sistemului este controlată exclusiv prin întreruperi și semnalizată vizual prin LED-uri.

```c
/*---------------------------------------------------------------------
 * Fișier: bootloader_app_switch.c
 * Utilizat pentru comutarea între modul aplicație și bootloader
 *---------------------------------------------------------------------*/

// Includes
#include <ioavr.h>
#include <intrinsics.h>

/*----------------------------------------------------------------------
 * Public defines
 *---------------------------------------------------------------------*/

// Butoane de switch mode

// Registrul de citire pentru butonul de schimbare mod
#define MODE_SWITCH_BUTTON_PIN_REG   PINB
// Registrul de direcție pentru portul butonului
#define MODE_SWITCH_BUTTON_DDR       DDRB
// Registrul de setare pentru portul butonului
#define MODE_SWITCH_BUTTON_PORT      PORTB
// Pinul fizic al butonului pe portul B
#define MODE_SWITCH_BUTTON_PIN       6
// Linia de întrerupere pin-change asociată butonului
#define MODE_SWITCH_BUTTON_PCINT     PCINT6

// Butoane de action

// Registrul de citire pentru butonul de acțiune
#define ACTION_BUTTON_PIN_REG        PINE
```

```
// Registrul de direcție pentru portul butonului
#define ACTION_BUTTON_DDR           DDRE
// Registrul de setare pentru portul butonului
#define ACTION_BUTTON_PORT          PORTE
// Pinul fizic al butonului pe portul E
#define ACTION_BUTTON_PIN           6
// Numărul întreruperii externe asociate butonului
#define ACTION_BUTTON_INT           INT6

// LED aplicație

// Registrul de direcție pentru LED-ul aplicație
#define LED_A_DDR   DDRA
// Registrul de setare pentru LED-ul aplicație
#define LED_A_PORT  PORTA
// Pinul fizic al LED-ului aplicație
#define LED_A_PIN   5

// LED bootloader
// Registrul de direcție pentru LED-ul bootloader
#define LED_B_DDR   DDRA
// Registrul de setare pentru LED-ul bootloader
#define LED_B_PORT  PORTA
// Pinul fizic al LED-ului bootloader
#define LED_B_PIN   6

// LED pentru indicarea modului

// Registrul de direcție pentru LED-ul indicator
#define LED_C_DDR   DDRA
// Registrul de setare pentru LED-ul indicator
#define LED_C_PORT  PORTA
// Pinul fizic al LED-ului indicator
#define LED_C_PIN   7

// Toggle LED-uri
#define LED_A_TOGGLE()     (LED_A_PORT ^= (1 << LED_A_PIN))
#define LED_B_TOGGLE()     (LED_B_PORT ^= (1 << LED_B_PIN))

// LED C ON/OFF
#define LED_C_ON()         (LED_C_PORT |= (1 << LED_C_PIN))
#define LED_C_OFF()        (LED_C_PORT &= ~(1 << LED_C_PIN))

// Flag pentru stocarea modului curent
volatile unsigned char in_bootloader_mode = 0;

// Handler pentru modul aplicație
#pragma vector = INT6_vect

__interrupt void Application_Action_ISR(void) {
    // Acțiune specifică aplicației: pulsează LED-ul A
    LED_A_TOGGLE();
    __delay_cycles(1600000);
    LED_A_TOGGLE();
}

// Handler pentru modul bootloader
__interrupt void Bootloader_Action_ISR(void) {
    // Acțiune specifică bootloader-ului: pulseaza LED-ul B
    LED_B_TOGGLE();
    __delay_cycles(1600000);
```

```
    LED_B_TOGGLE();
}

// Handler pentru comutarea modului
#pragma vector = PCINT0_vect

__interrupt void Mode_Switch_ISR(void) {
    __delay_cycles(160000); // 10 ms

    if ( !(MODE_SWITCH_BUTTON_PIN_REG & (1 << MODE_SWITCH_BUTTON_PIN)) )
{
        if (in_bootloader_mode) {

            // Se face trecerea la modul aplicație
            in_bootloader_mode = 0;
            LED_C_OFF(); // Se semnalizează intrarea în modul aplicație

            // Se activează permisiunea de a schimba IVSEL
            MCUCR = (1 << IVCE);
            // Se setează IVSEL la 0 (tabela la 0x0000)
            MCUCR = 0;

        } else {
            // Se face trecerea la modul bootloader
            in_bootloader_mode = 1;
            LED_C_ON(); // Se semnalizează intrarea în modul bootloader

            // Se activează permisiunea de a schimba IVSEL
            MCUCR = (1 << IVCE);
            // Se setează IVSEL la 1
            MCUCR = (1 << IVSEL);
        }

        // Se așteaptă eliberarea butonului
     while (!(MODE_SWITCH_BUTTON_PIN_REG & (1 <<
MODE_SWITCH_BUTTON_PIN)));
        __delay_cycles(160000);
    }
}

void main(void) {
    // Inițializare I/O
    LED_A_DDR |= (1 << LED_A_PIN);
    LED_B_DDR |= (1 << LED_B_PIN);
    LED_C_DDR |= (1 << LED_C_PIN);

    ACTION_BUTTON_DDR &= ~(1 << ACTION_BUTTON_PIN);
    // Activare pull-up
    ACTION_BUTTON_PORT |= (1 << ACTION_BUTTON_PIN);

    MODE_SWITCH_BUTTON_DDR &= ~(1 << MODE_SWITCH_BUTTON_PIN);
    // Activare pull-up
    MODE_SWITCH_BUTTON_PORT |= (1 << MODE_SWITCH_BUTTON_PIN);

    // Stare inițială
    LED_C_OFF();
    in_bootloader_mode = 0;

    // Se verifică folosirea tabelei la pornirea aplicației (IVSEL = 0)
    MCUCR = (1 << IVCE);
    MCUCR = 0;
```

```
    // Se configurează întreruperile
    // Generează întrerupere pe front descrescător pe INT6
    EICRB |= (1 << ISC61);
    EIMSK |= (1 << ACTION_BUTTON_INT); // Se activează întreruperea INT6
    // Se activează grupul de întreruperi pe pinii PCINT0:7
    PCICR |= (1 << PCIE0);
    // Se activează specific PCINT0
    PCMSK0 |= (1 << MODE_SWITCH_BUTTON_PCINT);
    // Se setează bitul I in SREG
    __enable_interrupt();

    while (1) {    }
}
```

bootloader_ivt.s90

Codul definește tabela de vectori de întrerupere pentru bootloader-ul microcontrolerului. La reset, saltul se face către rutina de inițializare C (**?C_STARTUP**), iar întreruperile hardware pentru butonul de acțiune și switch-ul de mod sunt redirecționate către ISR-urile specifice (**Bootloader_Action_ISR** și **Mode_Switch_ISR**). Practic, acest fișier configurează unde sare microcontrolerul pentru fiecare eveniment hardware în timpul bootloader-ului.

```
; Declararea simbolurilor externe care vor fi definite în codul C

; Punctul de intrare al programului principal (startup C)
EXTERN  ?C_STARTUP
; ISR pentru butonul de acțiune în bootloader
EXTERN  Bootloader_Action_ISR
; ISR pentru schimbarea modului
EXTERN  Mode_Switch_ISR

; Setarea segmentului pentru Vectorii de Întrerupere ai bootloader-ului

; Începerea IVT (Interrupt Vector Table) la adresa 0x1E000
ASEGN   BOOT_IVT_SEG, 0x1E000
; Vector de reset pentru bootloader
?RESET_vect_in_boot_ivt:
; Salt către rutina de startup C la reset
    JMP     ?C_STARTUP
; Vector pentru întreruperea butonului de acțiune
; Adresa corespunzătoare INT6 în IVT
ORG     0x001C
; Salt către ISR-ul Bootloader Action
    JMP     Bootloader_Action_ISR
; Vector pentru întreruperea switch-ului de mod
; Adresa corespunzătoare PCINT6 în IVT
ORG     0x0026
; Salt către ISR-ul Mode Switch
    JMP     Mode_Switch_ISR
END
```

4.11 BIBLIOGRAFIE

1. *"8-bit AVR Microcontroller ATmega 1280 Datasheet", Microchip Technology*
2. *"Schematic for UNI Clicker", Mikroe*
3. *"Schematic for ATmega1280: SiBrain", Mikroe*
4. *"Interrupt", Wikipedia*
5. *"Difference Between Hardware Interrupt and Software Interrupt", Geeks For Geeks*

5. COMUNICAREA SERIALĂ – UART

5.1 UNDE SE FOLOSEȘTE UART?

USART (**Universal Synchronous / Asynchronous Receiver / Transmitter**) este un modul folosit pe scară largă în sisteme embedded pentru comunicarea serială asincronă între microcontrolere sau între un microcontroler și periferice. Permite transmiterea și recepția de date pe un singur fir de date (**Rx / Tx**) fără ceas comun, folosind start / stop bits și, opțional, bit de paritate. Acesta este folosit pentru:

- **Plăci de dezvoltare:** trimiterea și recepția datelor între microcontroler și PC;
- **Module Bluetooth sau Wi-Fi:** transfer de comenzi și date;
- **Module GPS sau senzori externi:** recepția datelor în timp real;
- **Depanare și logging:** transmiterea mesajelor de stare către un terminal serial.

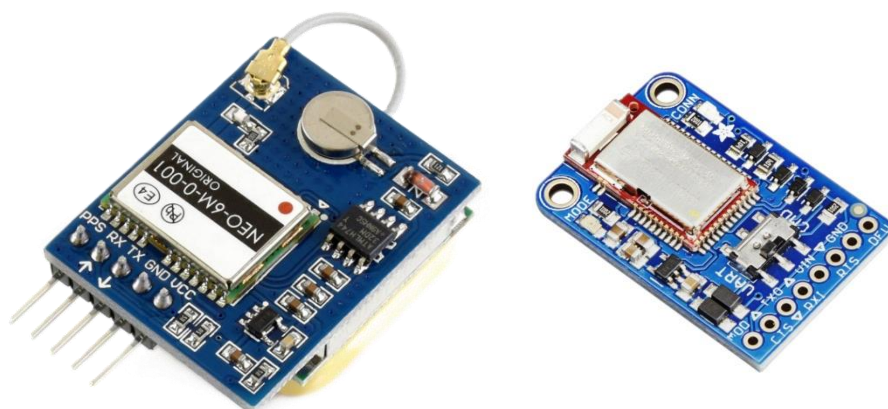


Figura 5.1 – Un modul UART GPS (stânga) și un modul UART Bluetooth BLE (dreapta)

5.2 CUNOȘTINȚE NECESARE

Pentru a putea parcurge acest capitol, este necesar să cunoașteți următoarele subiecte din capitolele anterioare, și nu numai:

- Noțiuni generale despre microcontrolere și funcționarea acestora;
- Cunoașterea arhitecturii microcontrolerului ATmega1280;
- Utilizarea kit-ului hardware Mikroe;
- Terminal serial (PuTTY).

5.3 ABSTRACT

Acest capitol are în vedere detalierea funcționării și utilizării modulului **UART** (**Universal Synchronous / Asynchronous Receiver-Transmitter**) pe microcontrolerul **ATmega1280**, explicând configurarea, programarea și aplicarea acestuia pentru a asigura comunicarea serială asincronă. Vom acoperi principiile de bază ale comunicației seriale, modelul comunicației seriale, tipurile de comunicație serială, controlul fluxului de date, conectorii și cablurile utilizate. De asemenea, vom include un studiu de caz pentru a demonstra o implementare practică a unei comunicații seriale de tip echo folosind polling pentru recepția și transmisia datelor.

5.4 HARDWARE ȘI SOFTWARE NECESAR

- ATmega1280 SiBRAIN;
- UNI Clicker;
- Atmel ICE;
- IAR *Embedded Workbench* IDE 7.30.5;
- Osciloscop.

5.5 INTRODUCERE ÎN COMUNICAȚIA SERIALĂ

Transmisia digitală de date a evoluat de la conexiunea între un calculator şi echipamentele periferice, la calculatoare care comunică în reţele internaţionale complexe. Însă sunt multe de învăţat pornind de la simpla legătură punct la punct sau RS232 după standardul EIA. Cu toate că transferul paralel este mai rapid, majoritatea transmisiilor de date între calculatoare sunt făcute pe cale serială pentru a reduce costul cablurilor şi conectorilor. Există şi limitări fizice de distanţă, care nu pot fi depăşite de magistrale paralele. În comunicaţia serială, datele sunt transmise bit cu bit.

Toate comunicaţiile sunt caracterizate de 3 elemente principale:
- **Date** - înţelegerea lor, scheme de codificare, cantitate;
- **Temporizări** - sincronizarea între receptor şi emiţător, frecvenţă şi fază;
- **Semnale** - tratarea erorilor, controlul fluxului şi rutarea interfeţelor seriale.

5.6 MODELUL COMUNICAȚIEI SERIALE

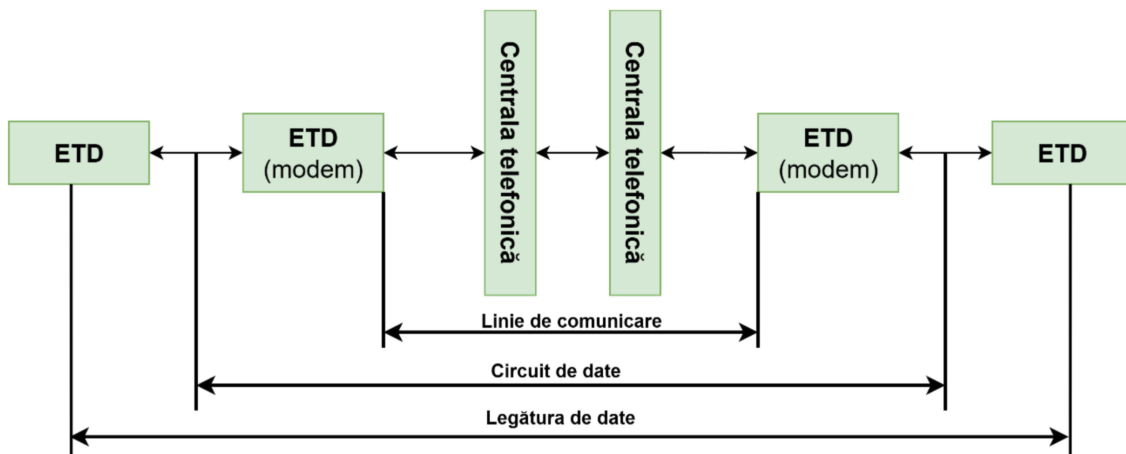


Figura 5.2 – Sistem de comunicație serială

Componentele unui sistem de comunicaţie serială sunt următoarele:
1. **ETD (echipamente terminale de date: calculatoare, terminale de date):** acestea conţin şi interfeţele seriale sau controlerele de comunicaţie.
2. **ECD (echipamente pentru comunicaţia de date):** aceste echipamente se numesc modem-uri şi permit calculatorului să transmită informaţii printr-o linie telefonică analogică.
   Funcţiile principale realizate de un modem sunt următoarele:
   - Conversia DA a informaţiilor din calculator şi AD a semnalelor de pe linia telefonică analogică;
   - Modularea / demodularea unui semnal purtător. La transmisie, modemul suprapune (modulează) semnalele digitale ale calculatorului peste semnalul purtător al liniei telefonice. La recepţie, modemul extrage (demodulează) informaţiile transportate de semnalul purtător şi le transferă calculatorului.
3. **Linia de comunicaţie:** reprezintă o linie fizică sau o linie telefonică. Linia telefonică poate fi o linie comutată (conectată la o centrală telefonică) sau o linie închiriată (dedicată).
4. **Circuitul de date:** cuprinde porţiunea dintre două echipamente terminale de date, modem-urile şi linia de comunicaţie. Pe distanţe reduse, este posibilă comunicaţia serială directă între două echipamente terminale de date prin linii fizice, fără utilizarea unor modem-uri. În acest caz, circuitul de date este reprezentat de aceste linii.
5. **Legătura de date:** conţine circuitul de date şi interfeţele seriale ale echipamentelor terminale de date.

În funcţie de numărul de echipamente interconectate, o legătură serială poate fi **punct la punct** (două echipamente) sau **multi-punct** (mai mult de două echipamente).

5.7 TIPURI DE COMUNICAȚII SERIALE

Din punctul de vedere al direcţiei de transfer, se pot distinge următoarele tipuri de comunicaţie serială:
- Simplex;
- Semiduplex;
- Duplex.

În cazul comunicaţiei **simplex**, datele sunt transferate întotdeauna în aceeaşi direcţie, de la echipamentul transmiţător la cel receptor. La comunicaţia **semiduplex**, fiecare echipament terminal de date funcţionează alternativ ca transmiţător, iar apoi ca receptor. Pentru acest tip de conexiune, este suficientă o singură linie de transmisie (două fire de legătură). Într-o comunicaţie **duplex** (numită şi **duplex integral**), datele se transferă simultan în ambele direcţii. Primele conexiuni duplex necesitau două linii de transmisie (patru fire de legătură), dar conexiunile ulterioare necesită o singură linie.

Din punctul de vedere al sincronizării dintre transmiţător şi receptor, există două tipuri de comunicaţie serială:
- Asincronă (**UART**);
- Sincronă (**USART**).

5.7.1 COMUNICAȚIA ASINCRONĂ – UART

Pentru a asigura sincronizarea dintre transmiţător şi receptor, fiecare caracter transmis este precedat de un bit de **START**, cu valoarea logică **0** (**“space”**) şi este urmat de cel puţin un bit de **STOP**, cu valoarea logică **1** (**“mark”**).

Biţii de **START** şi de **STOP** încadrează fiecare caracter transmis; caracterul transmis între aceşti 2 biţi reprezintă un cadru de date. Un asemenea cadru reprezintă informaţia digitală de bază într-un sistem de comunicaţie serială. În cazul comunicaţiei asincrone, intervalul de timp între transmisia a 2 caractere succesive este variabil, pe durata acestui interval linia de comunicaţie fiind în starea **1 logic**. Acest mod de comunicaţie este numit şi **START-STOP**.

Sincronizarea la nivel de bit se realizează cu ajutorul semnalelor de ceas locale cu aceeaşi frecvenţă. Atunci când receptorul detectează începutul unui caracter indicat prin bitul de **START**, porneşte un oscilator de ceas local, care permite eşantionarea corectă a biţilor individuali ai caracterului. Eşantionarea biţilor se realizează aproximativ la mijlocul intervalului corespunzător fiecărui bit.

Figura 5.3 ilustrează transmisia caracterului cu codul ASCII **0x61**. După bitul de **START**, având durata T corespunzătoare unui bit, transmisia caracterului începe cu bitul cel mai puţin semnificativ (**b0**). După transmisia bitului cel mai semnificativ (**b8**), se transmite un bit de paritate **p**; în acest exemplu, paritatea este **impară**. Bitul de paritate este opţional, iar în cazul în care se adaugă la caracterul transmis, paritatea poate fi selectată pentru a fi pară sau impară. Există şi posibilitatea ca bitul de paritate să fie setat la **0** sau **1**, indiferent de paritatea efectivă a caracterului. În exemplul ilustrat, la sfârşitul caracterului se transmit doi biţi de **STOP s1** şi **s2**, după care linia rămâne în starea **1 logic** un timp nedefinit. Acest timp corespunde unui interval de pauză.

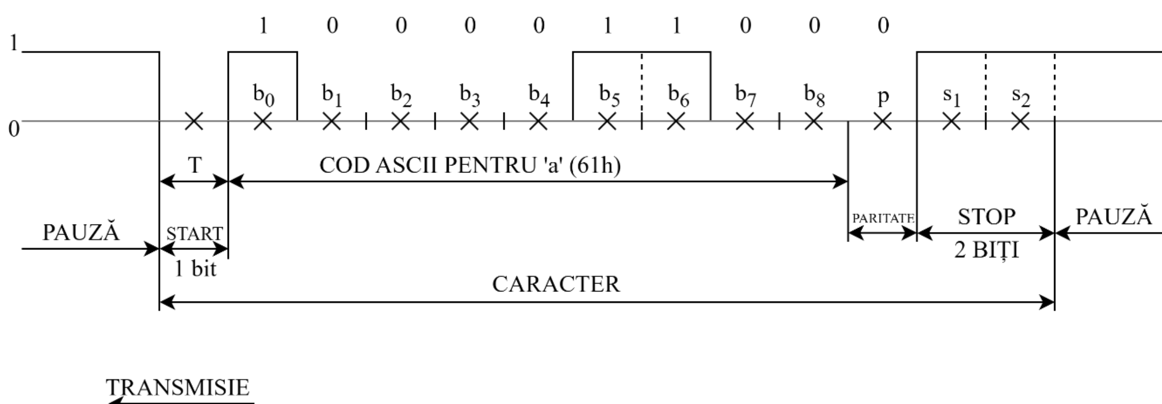


Figura 5.3 – Comunicația asincronă

În cazul comunicaţiei asincrone, sincronizarea la nivel de bit este asigurată numai pe durata transmisiei efective a fiecărui caracter. O asemenea comunicaţie este orientată pe caractere individuale şi are dezavantajul că necesită informaţii suplimentare în proporţie de cel puţin 25% pentru identificarea fiecărui caracter.

5.7.2 COMUNICAȚIA SINCRONĂ – USART

În cazul comunicaţiei sincrone, un cadru nu conţine un singur caracter, ci un **bloc de caractere** sau un mesaj. Sincronizarea la nivel de bit trebuie asigurată permanent, nu numai în timpul transmisiei propriu-zise, ci şi în intervalele de pauză. De aceea, timpul este divizat în mod continuu în intervale elementare la transmiţător, intervale care trebuie regăsite apoi la receptor. Aceasta pune anumite probleme: dacă ceasul local al receptorului are o frecvenţă care diferă într-o anumită măsură de frecvenţa transmiţătorului, vor apare erori la recunoaşterea caracterelor, din cauza lungimii blocurilor de caractere.

Pentru a se evita asemenea erori, ceasul receptorului trebuie resincronizat frecvent cu cel al transmiţătorului. Aceasta se poate realiza dacă se asigură că există suficiente tranziţii de la **1** la **0** şi de la **0** la **1** în mesajul transmis. Dacă datele de transmis constau din şiruri lungi de **1** sau de **0**, trebuie inserate tranziţii suficiente pentru resincronizarea ceasurilor. Asemenea tehnici sunt dificil de implementat, astfel încât se utilizează de obicei o tehnică numită **comunicaţie asincronă sincronizată** (numită în mod simplu comunicaţie sincronă).

Acest tip de comunicaţie este caracterizat de faptul că, deşi mesajul este transmis într-un mod sincron, nu există o sincronizare în intervalul de timp dintre două mesaje. Informaţia este transmisă sub forma unor blocuri de caractere sau a unor biţi succesivi, fără biţi de **START** şi **STOP**. Pentru ajustarea oscilatorului local la începutul unui mesaj, fiecare mesaj este precedat de un număr de caractere speciale de sincronizare, de exemplu, caracterul **SYN** (**0x16**). Pentru menţinerea sincronizării, se pot insera caractere de sincronizare suplimentare în mesajul transmis, la anumite intervale de timp.

La receptor există trei nivele de sincronizare:

- Sincronizare la nivel de bit, utilizând circuite cu calare de fază **PLL** (**Phase–Locked Loop**), pe baza tranziţiilor existente în semnalul recepţionat;
- Sincronizare la nivel de caracter, asigurată prin recunoaşterea anumitor caractere de sincronizare;
- Sincronizare la nivel de bloc sau mesaj, care depinde de protocolul de date utilizat.

5.7.3 STANDARDUL RS-232C

Specificaţiile electrice ale portului serial au fost definite în standardul **RS-232C** (**Reference Standard No. 232, Revision C**), elaborat în anul 1969 de către Comitetul de Standarde din SUA, cunoscut azi sub numele de **Asociaţia Industriei Electronice** (**EIA – Electronic Industries Association**). Standardul a fost elaborat pentru comunicaţia digitală între un calculator şi un terminal aflat la distanţă sau între două terminale fără utilizarea unui calculator. Terminalele erau conectate prin linii telefonice, astfel încât erau necesare modem-uri la ambele capete ale liniei de comunicaţie.

Standardul RS-232C a suferit diferite modificări, fiind elaborate mai multe revizii ale acestuia. De exemplu, în anul 1987 a fost elaborată o nouă revizie a standardului, numită **EIA RS-232D**. În anul 1991, **EIA** şi **Asociaţia Industriei de Telecomunicaţii** (**TIA – Telecommunications Industry Association**) au elaborat revizia **E** a standardului (**EIA/TIA RS-232E**). Revizia curentă este **EIA RS-232F**, publicată în anul 1997. Totuşi, indiferent de revizia acestuia, standardul este numit de cele mai multe ori **RS-232C** sau **RS-232**.

În Europa, versiunea echivalentă standardului RS-232C este **V.24**, elaborată de comitetul **CCITT** (**Comité Consultatif International pour Téléphonie et Télégraphie**). Denumirea acestui comitet a fost schimbată la începutul anilor 1990 în **International Telecommunications Union** (**ITU**). Ambele standarde specifică semnalele utilizate pentru comunicaţie, nivelele de tensiune, protocolul utilizat pentru controlul fluxului de date şi conectorii interfeţei seriale.

Standardul RS-232C defineşte atât o comunicaţie **asincronă**, cât şi una **sincronă**. Nu sunt definite detalii cum sunt codificarea caracterelor (**ASCII, Baudot, EBCDIC**), încadrarea caracterelor (lungimea caracterului, numărul biţilor de stop, paritatea) şi nici vitezele de comunicaţie, deşi standardul este destinat pentru viteze mai mici de **20.000 biţi/s**. Echipamentele actuale permit însă viteze superioare de comunicaţie, utilizând nivele de tensiune care sunt compatibile cu cele specificate de standard. Porturile seriale ale calculatoarelor permit, de obicei, selecţia uneia din următoarele viteze de comunicaţie: 150, 300, 600, 1.200, 2.400, 4.800, 9.600, 19.200, 38.400, 57.600, şi 115.200 biţi/s.

O legătură de bază **RS-232C** necesită doar 3 conexiuni: una pentru transmisie, una pentru recepţie şi una pentru masa electrică comună. Cele mai multe legături seriale utilizează însă şi semnale pentru controlul fluxului de date. Spre deosebire de alte tipuri de comunicaţie serială care sunt diferenţiale, comunicaţia **RS-232C** este una obişnuită, utilizând câte un fir pentru fiecare semnal. Deşi astfel se simplifică circuitele necesare interfeţei, în acelaşi timp se reduce şi distanţa maximă de comunicaţie în cazul unei legături directe, fără utilizarea modem-urilor. Standardul **RS-232C** specifică o distanţă maximă de 15 m. Distanţa poate fi mărită dacă se utilizează viteze de comunicaţie mai reduse. Tensiunile electrice specificate de standardul **RS-232C** sunt următoarele:

- Valoarea logică **0** corespunde unei tensiuni pozitive între **+3 V** şi **+25 V**;
- Valoarea logică **1** corespunde unei tensiuni negative între **–3 V** şi **–25 V**.

5.8 CONTROLUL FLUXULUI DE DATE

Pentru a fi posibilă comunicaţia între dispozitive cu viteze diferite, proiectanţii interfeţei seriale au prevăzut semnale speciale pentru controlul fluxului de date. Aceste semnale permit unui echipament oprirea şi apoi reluarea transmiterii datelor la cererea echipamentului de la celălalt capăt al liniei de comunicaţie serială. Pe lângă această metodă **hardware** pentru controlul fluxului de date, există şi o metodă **software**, bazată pe transmiterea unor caractere speciale între cele două echipamente. Atunci când echipamentul receptor (de exemplu, o imprimantă) nu mai poate primi date deoarece bufferul acestuia este plin, transmite un anumit caracter de control echipamentului transmiţător (de exemplu, calculatorului). Atunci când echipamentul receptor poate primi noi date, transmite un alt caracter de control care semnalează echipamentului transmiţător că poate relua transmiterea datelor.

De obicei, metoda de control care va fi utilizată de calculator poate fi selectată prin intermediul driver-ului software al controlerului serial. Unele programe pot utiliza în mod implicit o anumită metodă. În cazul perifericelor, metoda de control poate fi selectată fie prin program, fie printr-un comutator. Este important să se utilizeze aceeaşi metodă de control atât pentru calculator, cât şi pentru periferic pentru a evita pierderile de date.

5.8.1 METODA DE CONTROL HARDWARE

Metoda de control **hardware** presupune utilizarea unui protocol de comunicaţie cu ajutorul **semnalelor de control ale interfeţei seriale**. Protocolul utilizat se bazează pe comunicaţia serială prin intermediul unor modem-uri şi a unei linii telefonice, pentru care a fost elaborată interfaţa serială originală. Acest protocol implică stabilirea conexiunii între două modem-uri prin linia telefonică şi menţinerea fluxului de date dintre acestea cât timp conexiunea este activă. Etapele acestui protocol sunt descrise în continuare. Într-o formă simplificată, acest protocol este utilizat şi în cazul comunicaţiei seriale directe între două echipamente, fără utilizarea unor modem-uri şi a unei linii telefonice.

Atunci când un modem aflat la distanţă doreşte stabilirea conexiunii cu modemul local, transmite semnalul de apel pe linia telefonică. Acest semnal este detectat de către modemul local, care activează semnalul **RI** pentru a informa calculatorul local asupra existenţei unui apel telefonic.

La detectarea activării semnalului **RI**, pe calculatorul local se lansează în execuţie un program de comunicaţie. Acest program indică disponibilitatea calculatorului de a comunica prin activarea semnalului **DTR**.

Atunci când modemul local sesizează faptul că terminalul de date (calculatorul) este pregătit, răspunde la apelul telefonic şi aşteaptă activarea semnalului purtător de către modemul aflat la distanţă. Atunci când modemul local detectează semnalul purtător, activează semnalul **CD**.

Modemul local negociază cu modemul aflat la distanţă o conexiune cu anumiţi parametri. De exemplu, cele două modem-uri pot determina viteza optimă de comunicaţie în funcţie de calitatea legăturii telefonice. După această negociere, modemul local activează semnalul **DSR**.

La sesizarea activării semnalului **DSR**, programul de pe calculatorul local activează semnalul **RTS** pentru a indica modemului că poate transmite date către calculator.

Atunci când modemul sesizează activarea semnalului **RTS**, activează semnalul **CTS** pentru a indica faptul că este pregătit pentru recepţia datelor de la calculator.

Apoi, datele sunt transferate în ambele sensuri între echipamentele aflate la distanţă, pe liniile **TD** şi **RD**.

Deoarece viteza liniei telefonice este mai redusă decât cea a legăturii dintre calculator şi modemul local, bufferul modemului se va umple. Modemul local solicită calculatorului oprirea transmiterii datelor prin dezactivarea semnalului **CTS**. La golirea bufferului, modemul reactivează semnalul **CTS**.

În cazul în care calculatorul nu mai poate primi date de la modem, dezactivează semnalul **RTS**. Atunci când calculatorul poate primi din nou date de la modem, reactivează semnalul **RTS**.

La încheierea sesiunii de comunicaţie, semnalul purtător este dezactivat, iar modemul local dezactivează semnalele **CD**, **CTS** şi **DSR**.

Atunci când sesizează dezactivarea semnalului **CD**, calculatorul local dezactivează semnalele **RTS** şi **DTR**.

Din protocolul descris mai sus, rezultă următoarele:

- Calculatorul trebuie să detecteze activarea semnalelor **DSR** şi **CTS** înainte de a transmite date către modem. Dezactivarea oricăruia din aceste semnale va opri, de obicei, fluxul de date de la calculator;
- Modemul trebuie să detecteze activarea semnalelor **DTR** şi **RTS** înainte de a transmite date pe lini serială sau către calculator. Dezactivarea semnalului **DTR** va opri transmiterea datelor pe linia serială, iar dezactivarea semnalului **RTS** va opri transmiterea datelor către calculator;
- Starea semnalului **CD** nu este interpretată de toate sistemele de comunicaţie serială. La anumite sisteme, semnalul **CD** trebuie să fie activat înainte ca terminalul de date să înceapă transmiterea datelor. La alte sisteme, starea semnalului **CD** este ignorată.

5.8.2 METODA DE CONTROL SOFTWARE

Metoda **software** pentru controlul fluxului de date presupune transmiterea unor **caractere de control** între cele două echipamente. De exemplu, perifericul va transmite un anumit caracter de control pentru a indica faptul că nu mai poate primi date de la calculator şi va transmite un alt caracter de control pentru a indica faptul că transmiterea datelor poate fi reluată de calculator. Există două variante ale acestei metode. Prima variantă utilizează caracterele de control **XON / XOFF**, iar a doua variantă utilizează caracterele de control **ETX / ACK**.

În cazul utilizării variantei **XON / XOFF**, perifericul transmite caracterul **XOFF** pentru a indica faptul că bufferul său este plin şi transmiterea datelor trebuie oprită de calculator. Acest caracter mai este denumit **DC1** (**Device Control 1**) şi are codul ASCII **0x13**, fiind echivalent cu caracterul **Ctrl + S**. Caracterul **Ctrl + S** poate fi introdus şi de utilizator la anumite programe de comunicaţie pentru a opri transmiterea datelor de către un echipament cu care este conectat calculatorul.

Atunci când perifericul este pregătit pentru a primi noi date, transmite calculatorului caracterul **XON**. Acesta mai este denumit **DC3** (**Device Control 3**) şi are codul ASCII **0x11**, fiind echivalent cu caracterul **Ctrl + Q**. La anumite programe de comunicaţie, introducerea caracterului **Ctrl + Q** anulează efectul caracterului **Ctrl + S**.

În cazul utilizării variantei **ETX / ACK**, transmiterea caracterului **ETX** (**End of TeXt**) de către periferic indică faptul că transmiterea datelor trebuie oprită de calculator. Acest caracter are codul ASCII **0x03** şi este echivalent cu caracterul **Ctrl + C**. Transmiterea caracterului **ACK** (**ACKnowledge**) indică posibilitatea reluării transmiterii datelor de către calculator. Acest caracter are codul ASCII **0x06** şi este echivalent cu caracterul **Ctrl + F**.

5.9 CONECTORI

Porturile seriale pot utiliza unul din două tipuri de conectori. Conectorul **DB-25** cu 25 de pini a fost utilizat la calculatoarele din generaţiile anterioare. La calculatoarele mai noi se utilizează conectorul **DB-9** cu 9 pini. Pentru porturile seriale ale calculatoarelor se utilizează conectori tată, iar pentru porturile seriale ale echipamentelor periferice se utilizează conectori mamă.

Conectorul **DB-25** al portului serial are o formă similară cu conectorul **DB-25** al portului paralel. Portul serial care utilizează un conector **DB-25** se poate deosebi de portul paralel prin faptul că pentru portul serial se utilizează un conector tată, în timp ce pentru portul paralel se utilizează un conector mamă. **Figura 1.4** ilustrează conectorul DB-25 al portului serial.

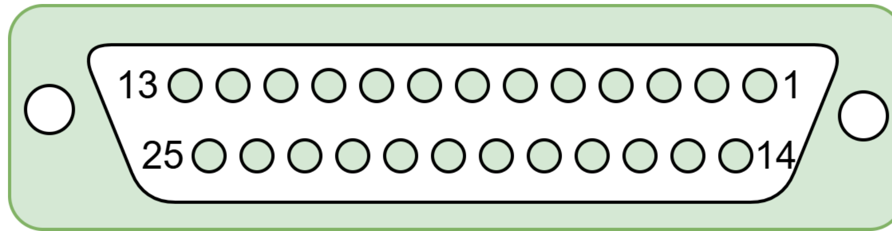


Figura 5.4 – Conectorul DB-25 male (tată)

Din cele 25 de semnale ale conectorului **DB-25**, se utilizează cel mult 10 semnale pentru o conexiune serială obişnuită. **Tabelul 5.1** indică numele acestor semnale şi asignarea lor la pinii conectorului DB-25.

Pin	Semnal	Semnificaţie	← In / Out →
1	PG	Protective Ground	←
2	TD	Transmit Data	←
3	RD	Receive Data	→
4	RTS	Request To Send	→
5	CTS	Clear To Send	
6	DSR	Data Set Ready	
7	SG	Signal Ground	←
8	DCD	Data Carrier Detect	→
20	DTR	Data Terminal Ready	←
22	RI	Ring Indicator	←

Tabelul 5.1 – Semnalele portului

Pentru a se reduce spaţiul ocupat de conectorul portului serial, conectorul DB-25 a fost înlocuit cu un conector de dimensiuni mai reduse, şi anume conectorul cu 9 pini **DB-9** (Figura 1.5).

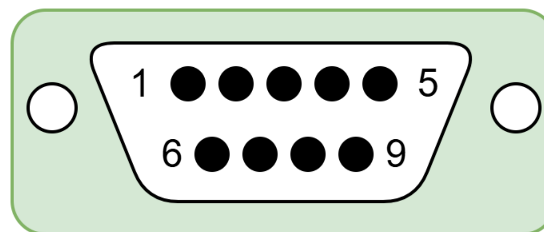


Figura 5.5 – Conectorul DB-9 female (mamă)

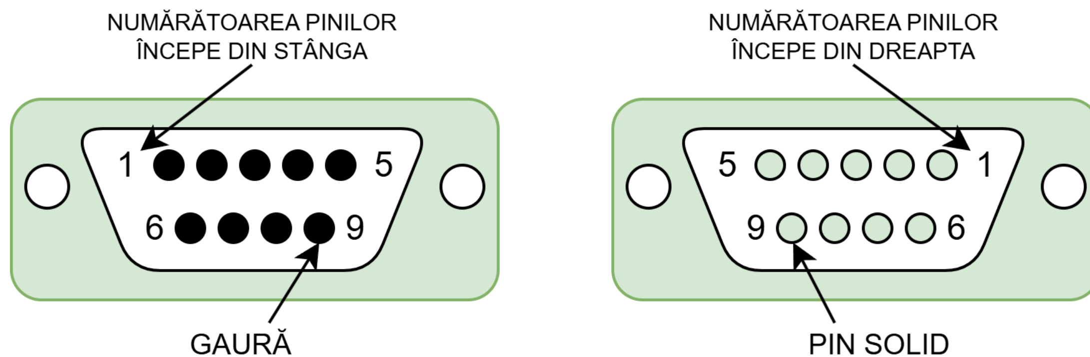


Figura 5.6 – Dierenţele între conectorul mamă (stânga) şi conectorul tată (dreapta)

5.10 CABLURI

Există mai multe variante de cabluri care se pot utiliza pentru comunicaţia serială. Pentru viteze de comunicaţie reduse şi lungimi scurte, se pot utiliza cabluri obişnuite, care nu sunt ecranate. Pentru a reduce interferenţele cu alte echipamente, trebuie utilizate cabluri ecranate care conţin un înveliş sub forma unei folii de aluminiu. În mod ideal, ecranul cablului trebuie conectat la masa de protecţie a conectorului, dacă acesta este de tip **DB-25**. Conectorul **DB-9** nu conţine un pin pentru masa de protecţie. În cazul utilizării conectorilor de acest tip, ecranul cablului se poate conecta la masa electrică.

Observații: În cazul cablului serial care utilizează conectori **DB-25**, masa electrică sau masa de semnal **SG** (**Signal Ground**) este separată de masa mecanică sau masa de protecţie **PG** (**Protective Ground**). Masa de protecţie este conectată direct la carcasa conectorului (şi a echipamentului), având un rol de protecţie. Prin realizarea acestei conexiuni, carcasele metalice ale celor două echipamente conectate prin cablul serial se vor afla la acelaşi potenţial, evitându-se formarea unor diferenţe de tensiune între cele două echipamente, tensiuni care pot fi periculoase pentru acestea. Deseori, conexiunea mesei de protecţie lipseşte din cablurile seriale. Masa de protecţie **PG** nu trebuie conectată niciodată la masa electrică **SG**.

Semnalele interfeţei seriale au fost prevăzute în scopul conectării unui echipament terminal de date (**ETD**) la un echipament pentru comunicaţia de date (**ECD**). Atunci când se conectează două asemenea echipamente, de exemplu, un calculator cu un modem, care dispun de conectori de acelaşi tip (de exemplu, **DB-25**), este necesar un cablu care conectează pinii cu acelaşi număr ai conectorilor de la cele două capete. Acesta este un cablu direct. Dacă se conectează două echipamente cu conectori diferiţi, este necesar un cablu adaptor. Dacă se conectează două echipamente terminale de date, de exemplu, două calculatoare, datele transmise pe pinul **TD** al unui echipament trebuie recepţionate pe pinul **RD** al celuilalt echipament. De aceea, conexiunile acestor pini trebuie inversate la cele două capete ale cablului; un asemenea cablu este numit **cablu inversor**.

5.10.1 CABLURI STRAIGHT–THROUGH

Cablul serial **straight-through** este utilizat pentru conectarea unui echipament **DTE** (ex. calculator) la un echipament **DCE** (ex. modem), păstrând semnalele în aceeași ordine logică. În acest tip de cablu, firele sunt conectate direct între pini corespunzători (ex. **TX** → **TX**, **RX** → **RX**), fără inversare. Este potrivit pentru comunicații standard **RS-232** unde dispozitivele au roluri complementare.

Mai jos sunt prezentate diagramele acestui tip de conexiune:

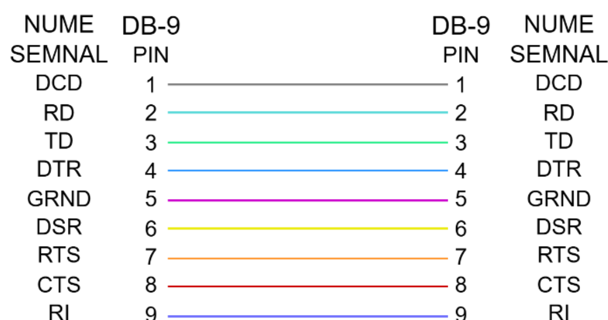


Figura 5.7 – Diagramă conexiune DB-9 : DB-9 straight-through

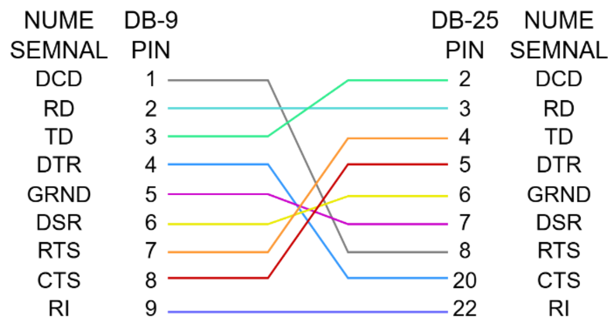


Figura 5.8 – Diagramă conexiune DB-9 : DB-25 straight-through

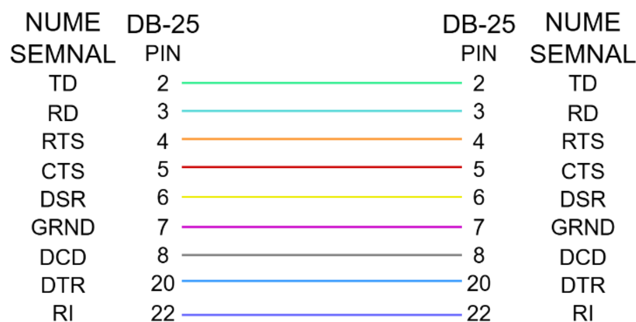


Figura 5.9 – Diagramă conexiune DB-25 : DB-25 straight-through

5.10.2 CABLURI NULL MODEM

Cablul serial **null modem** este folosit pentru conectarea directă între două echipamente de tip **DTE** (ex. două calculatoare), fără un modem intermediar. În acest caz, semnalele de transmisie și recepție (**TX** și **RX**), precum și liniile de control (**RTS / CTS** sau **DSR / DTR**), sunt încrucișate pentru a permite comunicarea bidirecțională. Acest tip de cablu simulează comportamentul unui modem prin cablare.

Mai jos sunt prezentate diagramele acestui tip de conexiune:

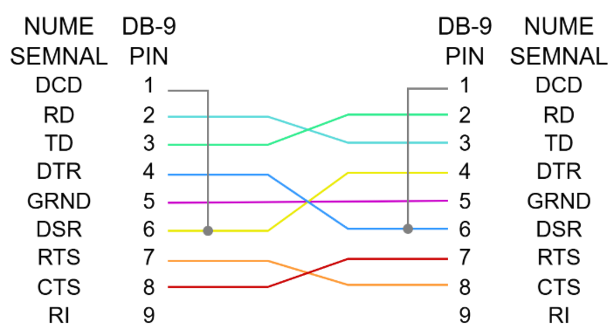


Figura 5.10 – Diagramă conexiune DB-9 : DB-9 null modem

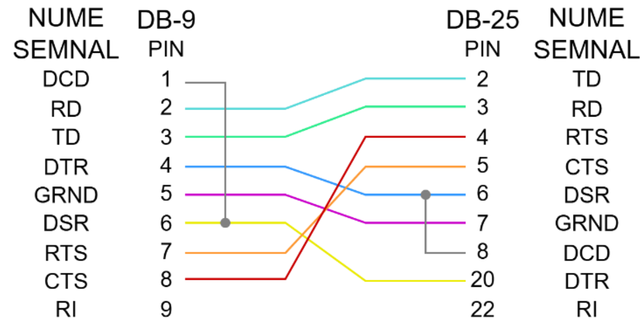


Figura 5.11 – Diagramă conexiune DB-9 : DB-25 null modem

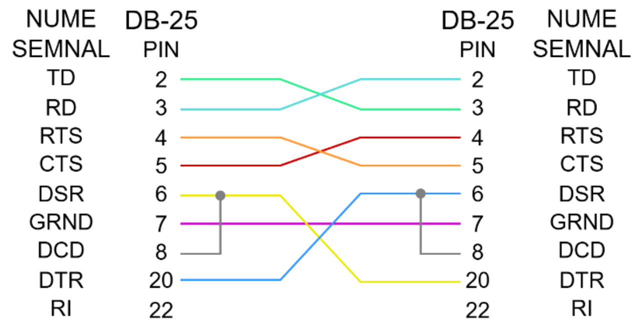


Figura 5.12 – Diagramă conexiune DB-25 : DB-25 null modem

5.11 MODULUL USART

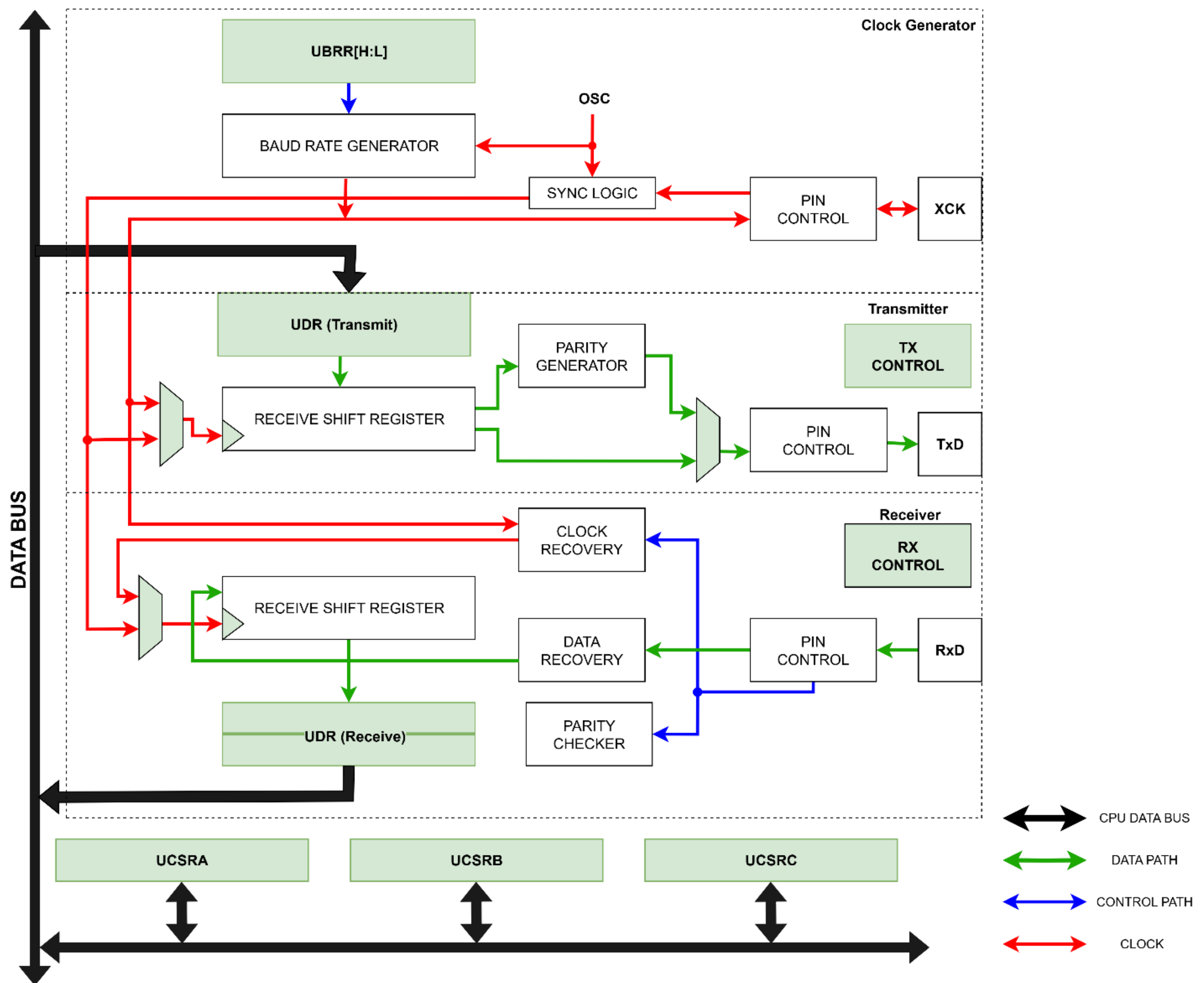


Figura 5.13 - Diagrama bloc USART

ATmega1280 dispune de 3 subsisteme pentru comunicaţia serială:
1. Universal Synchronous & Asynchronous Serial Receiver & Transmitter (**UART / USART**);
2. Serial Peripheral Interface (**SPI**);
3. Two-wire Serial Interface (**TWI**).

5.11.1 INTRODUCERE

USART este un periferic hardware foarte flexibil, folosit pentru **comunicarea serială** (transmiterea datelor bit cu bit pe o singură linie). Poate funcționa în două moduri principale:

- **Modul asincron (Asynchronous)**
- **Modul sincron (Synchronous)**: funcționează ca **Master** (generează CLK-ul) sau **Slave** (primește).

Microcontrolerele ATmega1280 au **4 module USART identice** (USART0, USART1, USART2, USART3), fiecare cu propriul set de regiștri.

Caracteristici principale:

- Poate trimite și primi date în același timp deoarece are buffere separate pentru transmisie și recepție
- Poate genera și verifica automat biți de paritate
- Viteza dublă: în modul asincron, viteza de transfer poate fi dublată
- Poate genera întreruperi pentru 3 evenimente distincte: RX complete, TX complete sau Data Register Empty.

Modulul USART este format din 3 părți principale: **generatorul de clock**, **transmițătorul** și **receptorul**.

5.11.2 GENERAREA CLOCK-ULUI

Logica de generare a clock-ului furnizează semnalul de bază pentru **Transmițător** și **Receptor**. Viteza de transfer este cunoscută ca **baud rate** și se măsoară în **biți pe secundă** (**bps**).

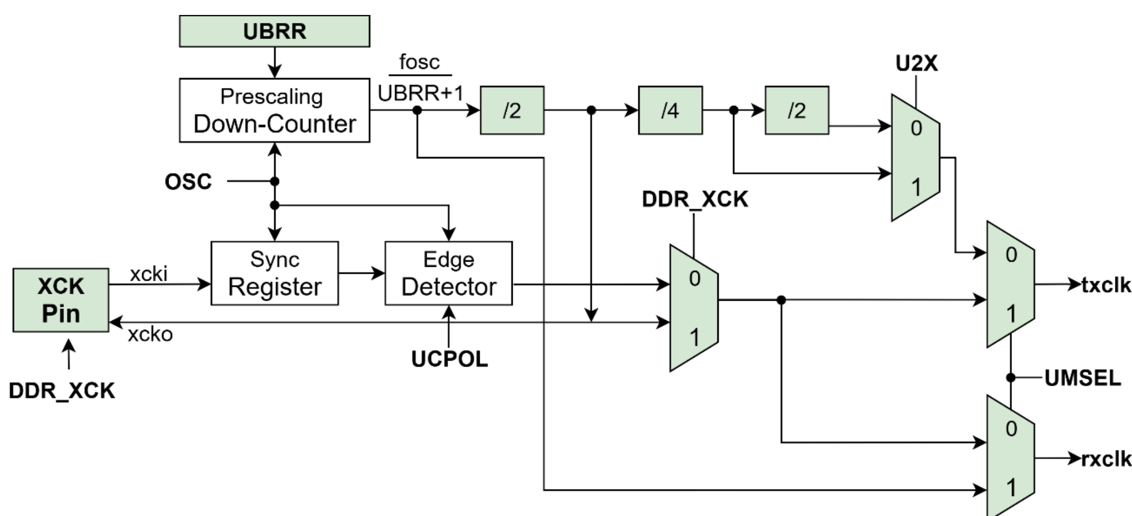


Figura 5.14 – Diagrama bloc pentru logica generării clock-ului

Clock-ul transmițătorului este divizat în **2**, **8** sau **16** stări depinzând de modul de operare utilizat. În funcție de modul de operare dorit trebuie calculată valoarea **Baud Rate-ului** și a **UBRR** conform tabelului următor:

Mod operare	Ecuația pentru Baud Rate	Ecuația pentru UBRR
Modul Asincron Normal (U2Xn = 0)	BAUD $= \frac{fosc}{16(UBRRn+1)}$	UBRRn $= \frac{fosc}{16(BAUD)}$ - 1
Modul Asincron Viteză Dublă (U2Xn = 1)	BAUD $= \frac{fosc}{8(UBRRn+1)}$	UBRRn $= \frac{fosc}{8(BAUD)}$ - 1
Modul Sincron	BAUD $= \frac{fosc}{2(UBBRn+1)}$	UBBRn $= \frac{fosc}{2(BAUD)}$ - 1

Tabelul 5.2 – Formulele de calcul ale Baud Rate-ului și UBRR-ului

5.11.3 FORMAT FRAME-URI

Definiție

Un frame serial reprezintă un caracter transmis, împreună cu biții de control. Acesta conține:

1. **Un bit de START: întotdeauna 0 logic.**
2. **Biți de date: între 5 și 9 biți.**
3. **Un bit de paritate (opțional): Pentru detecția erorilor.**
4. **Biți de STOP: Unul sau doi biți, întotdeauna 1 logic.**

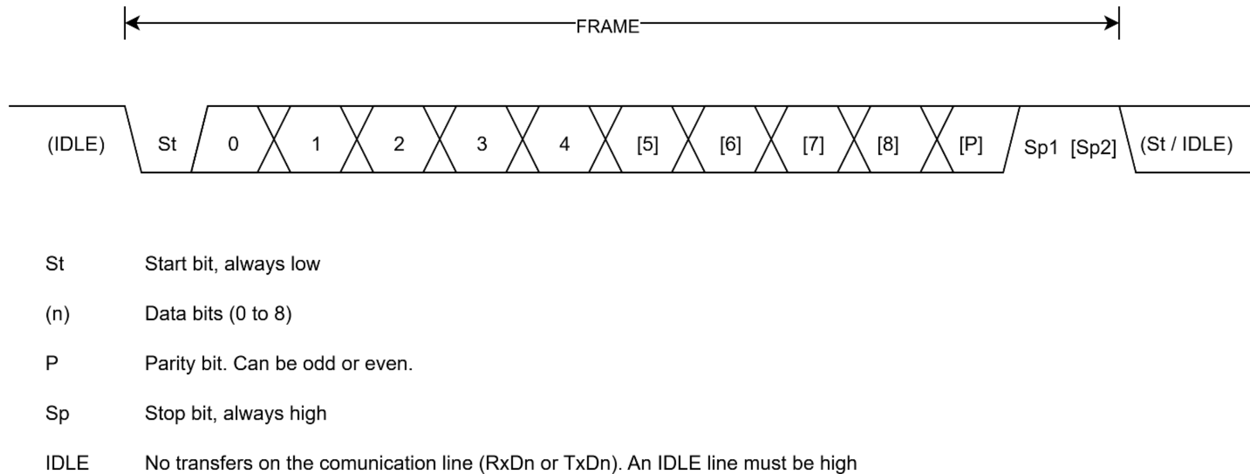


Figura 5.15 - Frame Format

5.11.4 INIȚIALIZAREA USART

Pentru a inițializa modulul **USART** trebuie urmați următorii pași:

1. Setarea **Baud Rate-ului**: se calculează valoarea **UBRR** și se încarcă registrele **UBRRH** și **UBRRL**.
2. Setarea **Formatului de Cadru**: se configurează numărul de biți de date, biți de paritate și biții de stop în registrul **UCSRC**.
3. Activarea Transmițătorului și / sau a Receptorului: se setează biții **TXEN** și / sau **RXEN** din reg. **UCSRB**.

5.12 DESCRIEREA REGIȘTRILOR

5.12.1 UDR – USART I/O DATA REGISTER

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDRn (Read)
	TXB[7:0]								UDRn (Write)
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 5.16 – Registrul UDR

Este un registru pe 8 biți care servește ca buffer atât pentru transmisie, cât și pentru recepție:

- La scriere, datele sunt puse în buffer-ul de transmisie (**TXB**).
- La citire, se accesează datele din buffer-ul de recepție (**RXB**).

5.12.2 UCSRNA – USART CONTROL AND STATUS REGISTER A

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial value	0	0	1	0	0	0	0	0	

Figura 5.17 – Registrul UCSRnA

- **RXCn (Bitul 7): Receive Complete**
  Acest flag este 1 când în buffer-ul de receptive există date necitite. Poate genera o întrerupere.

- **TXCn (Bitul 6): Transmit Complete**
  Acest flag este 1 când un frame a fost transmis și buffer-ul de transmisie este gol. Poate genera o întrerupere.

- **UDREn (Bitul 5): USART Data Register Empty**
  Acest flag este **1** când buffer-ul de transmisie este gol și gata să primească un nou caracter pentru a fi transmis. Poate genera o întrerupere.

- **FEn (Bitul 4): Frame Error**
  Setat pe **1** dacă este detectată o eroare la un frame (ex: bitul de stop este 0).

- **DORn (Bitul 3): Data OverRun**
  Setat pe **1** dacă o suprascriere de date are loc în buffer-ul de receptive.

- **UPEn (Bitul 2): Parity Error**
  Setat pe **1** dacă este detectată o eroare de paritate.

- **U2Xn (Bitul 1): Double the USART Transmission Speed**
  Când este setat pe **1** se dublează viteza de transfer în modul asincron.

- **MPCMn (Bitul 0): Multi-Processor Comunication Mode**
  Activează modul de comunicare multi-procesor.

5.12.3 UCSRNB – USART CONTROL AND STATUS REGISTER B

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial value	0	0	0	0	0	0	0	0	

Figura 5.18 – Registrul UCSRnB

- **RXCIEn (Bit 7): RX Complete Interrupt Enable**
  Activează întreruperea pe flag-ul **RXC**.

- **TXCIEn (Bit 6): TX Complete Interrupt Enable**
  Activează întreruperea pe flag-ul **TXC**.

- **UDRIEn (Bit 5): USART Data Register Empty Interrupt Enable**
  Activează întreruperea pe flag-ul **UDRE**.

- **RXENn (Bit 4): Receiver Enable**
  Activează receptorul **USART**.

- **TXENn (Bit 3): Transmitter Enable**
  Activează transmițătorul **USART**.

- **UCSZn2 (Bit 2): Character Size**
  Folosit împreună cu biții **UCSZ1:0** pentru a seta numărul de biți de date (5-9).

- **RXB8n (Bit 1): Receive Data Bit 8**
  Al 9-lea bit de date recepționat (în modul 9 biți de date).

- **TXB8 (Bit 0): Transmit Date Bit 8**
  Al 9-lea bit de date de transmis (în modul 9 biți de date).

5.13 UCSRNC – USART CONTROL AND STATUS REGISTER C

Bit	7	6	5	4	3	2	1	0	
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
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	1	1	0	

Figura 5.19 – Registrul UCSRnC

- **UMSELn1:0 (Bit 7:6): USART Mode Select**
  Setează modul de operare după următorul tabel:

UMSELn1	UMSELn0	Mode
0	0	USART asincron
0	1	USART sincron
1	0	Reserved
1	1	Master SPI

Tabelul 5.3 – Configurarea modului USART

- **UPMn1:0 (Bit 5:4): Parity Mode**
  Setează modul de paritate după următorul tabel:

UPMn1	UPMn0	Mode
0	0	Disabled
0	1	Reserved
1	0	Enabled, Even Parity
1	1	Enabled, Odd Parity

Tabelul 5.4 – Configurarea modului de paritate

- **USBSn (Bit 3): Stop Bit Select**
  Setează numărul de biți de stop:

USBSn	Stop Bit(s)
0	1-bit
1	2-bit

Tabelul 5.5 – Configurarea numărului de biți de stop

- **UCSZn1:0 (Bit 2:1): Character Size**
  Folosiți împreună cu UCSZ2 pentru a seta numărul de biți de date:

UCSZn2	UCSZn1	UCSZn0	Character size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Reserved
1	1	1	9-bit

Tabelul 5.6 – Configurarea numărului de biți de date

- **UCPOLn (Bit 0): Clock Polarity**
  Folosit doar în modul sincron pentru a defini polaritatea clock-ului:

UCPOLn	Transmited Data Changed (output of TxDn Pin)	Received Data Sampled (input on RxDn Pin)
0	Rising XCKn Edge	Falling XCKn Edge
1	Falling XCKn Edge	Rising XCKn Edge

Tabelul 5.7 – Configurarea parității clock-ului

5.14 PROBLEME

5.14.1 APLICAȚIE ECHO

Se cere implementarea unei aplicații software care realizează funcționalitatea de **echo** prin intermediul unei comunicații seriale **USART**. Aplicația trebuie să primească un **caracter transmis** de la PC sau alt dispozitiv conectat pe interfața serială și să îl retransmită imediat către sursă, asigurând astfel afișarea lui înapoi pe terminal.

Sugestii: *Configurarea presupune inițializarea vitezei de comunicație (**baud rate**) conform formulei standard, activarea receptorului și transmițătorului serial și configurarea pinilor corespunzători **TX** și **RX** (**TX** ca **ieșire**, **RX** ca **intrare**). Transmiterea și recepția datelor se realizează în mod blocant, prin verificarea flag-urilor din registrele de stare: pentru transmitere, se așteaptă eliberarea buffer-ului de trimitere (**UDRE3 = 1**), iar pentru recepție, se așteaptă sosirea unui caracter (**RXC3 = 1**). Astfel, fiecare caracter primit este preluat din registrul **UDR3** și retransmis imediat prin același registru.*

Conectări hardware: *Conectarea se face pe portul **microBUS 3**, unde pinii **TXD** și **RXD** ai microcontrolerului sunt legați la interfața de comunicație serială a plăcii. În cazul testării cu PC-ul, se poate folosi un cablu **USB-UART** pentru a deschide un terminal serial la **9600 baud**. Conectarea se poate face, la voia studentului, pe orice alt microBUS, cu condiția modificării pinilor aferenți microBUS-ului respectiv.*

`echo_usart.h`

Header-ul **`usart.h`** conține definiții pentru configurația hardware precum frecvența oscilatorului și viteza de comunicare dorită, declararea funcțiilor publice care pot fi folosite în alte părți ale programului și macro-uri pentru a simplifica configurarea pinilor și a regiștrilor.

```
/*---------------------------------------------------------------------------
 * Fișier: echo_usart.h
 * Utilizat pentru declararea funcțiilor necesare comunicării USART
 *-------------------------------------------------------------------------*/

#ifndef __ECHO_USART__
#define __ECHO_USART__

/*---------------------------------------------------------------------------
 * Includes
 *-------------------------------------------------------------------------*/

// General
#include <stdint.h>
#include <inavr.h>
#include <ioavr.h>

/*---------------------------------------------------------------------------
 * Public defines
 *-------------------------------------------------------------------------*/

// Frecvența oscilatorului
#define F_OSC 16000000UL

// Baud Rate
#define BAUD 9600
#define BAUD_RATE (F_OSC / 16 / BAUD - 1)


// Transmițătorul
#define TRANSMITTER (1 << TXEN3)

// Receptorul
#define RECEIVER (1 << RXEN3)

// Folosim pinul TX ca ieșire
#define TXD_OUT() DDRJ |= (1 << PJ1)

// Folosim pinul RX ca intrare
#define RXD_IN() DDRJ &= ~(1 << PJ0)

/*---------------------------------------------------------------------------
 * Public (exported) functions
 *-------------------------------------------------------------------------*/

// Funcția de inițializare a modulului USART
void usart_initialize(uint16_t baud_rate);

// Funcția de transmitere USART
void usart_transmit(uint8_t data);

// Funcția de recepție UART
uint8_t usart_receive(void);

#endif
```

echo_usart.c

Fișierul **echo_usart.c** are rolul de a implementa funcțiile din biblioteca **echo_usart.h**: **uart_initialize()** configurează viteza de comunicare și activeaza modulele de transmisie și recepție, **uart_transmit()** implementează o bucla de așteptare care se încheie doar atunci când registrul de transmisie este gol și ulterior scrie octetul pentru a fi trimis, iar **uart_receive()** așteaptă într-o buclă **while** până când un octet este recepționat complet. Odată ce datele sunt disponibile le returnează.

```
/*--------------------------------------------------------------------------
 * Fișier: echo_usart.c
 * Utilizat pentru definirea funcțiilor declarate anterior în uart.h
 *------------------------------------------------------------------------*/

/*--------------------------------------------------------------------------
 * Includes
 *------------------------------------------------------------------------*/

// General
#include "uart.h"

/*--------------------------------------------------------------------------
 * Public functions
 *------------------------------------------------------------------------*/

/*
 * Funcția configurează Baud Rate-ul pentru USART3, activează receptorul
 * și transmițătorul, iar apoi setează pinul TX ca ieșire și pinul RX ca
 * intrare.
 */
void usart_initialize(uint16_t baud_rate)
{
   // Se configurează Baud Rate-ul
   UBRR3H = (uint8_t)(baud_rate >> 8);
   UBRR3L = (uint8_t)(baud_rate & 0xFF);

   // Se pornește transmițătorul și receptorul
   UCSR3B = TRANSMITTER | RECEIVER;
   // Se setează pinul TXD ca ieșire
   TXD_OUT();
   // Se setează pinul RXD ca intrare
   RXD_IN();
}

/*
 * Funcția așteaptă până când buffer-ul de transmisie este gol (UDRE3 = 1)
 * iar apoi scrie un octet în registrul UDR3 pentru a fi trimis pe linia
 * serială.
 */
void usart_transmit(uint8_t data)
{
   while(!(UCSR3A & (1 << UDRE3)))
   {
     // Se așteaptă până când buffer-ul de transmisie e gol
   }
   UDR3 = data;
}
```

```
/*
 * Funcția așteaptă până când este recepționat un caracter (RXC3 = 1), iar
 * apoi returnează octetul citit din registrul UDR3.
 */
uint8_t usart_receive(void)
{
    while(!(UCSR3A &(1 << RXC3)))
    {
      // Se așteaptă recepționarea unui caracter
    }
    return UDR3;
}
```

main.c

Fișierul **main.c** reprezintă punctul de lansare în execuție. Se inițializează modulul **USART** (apelând **uart_initialize()**), după care se intră într-o buclă **while** infinită în care programul așteaptă să primească un caracter pe care apoi îl trimite.

```
/*------------------------------------------------------------------------
 * Fișier: main.c
 * Utilizat pentru lansarea în execuție a aplicației "ecou"
 *----------------------------------------------------------------------*/

// Includes
#include "usart.h"

void main(void)
{
  uint8_t aux;

  // Se inițializează modulul USART cu Baud Rate-ul predefinit
  usart_initialize(BAUD_RATE);

  // Buclă infinită pentru blocarea procesorului și crearea efectului de ecou
  while(1)
  {
   // Se primește tasta apăsată de la interfața serială
   aux = usart_receive();
   // Se trimite înapoi către sursă tasta primită, simulând un "ecou"
   usart_transmit(aux);
  }
}
```

5.14.2 APLICAȚIE SERIALĂ NON-BLOCANTĂ

Se cere implementarea unei aplicații software care permite transmiterea și recepția de date prin intermediul unei comunicații seriale **USART**, utilizând un buffer circular pentru transmitere și recepție. Aplicația trebuie să fie capabilă să trimită caractere individuale sau șiruri de caractere fără a bloca execuția programului principal și să preia caracterele primite, stocându-le temporar în buffer pentru prelucrare ulterioară.

***Sugestii**: Configurarea presupune inițializarea vitezei de comunicație (**baud rate**) conform formulei standard, activarea transmițătorului și receptorului serial, precum și configurarea piniilor corespunzători **TX** și **RX** (**TX** ca **ieșire**, **RX** ca **intrare**). Funcționalitatea de transmitere și recepție se realizează prin intermediul întreruperilor:*

- ***Transmitere** – la eliberarea registrului de date, următorul caracter din buffer este trimis automat.*
- ***Recepție** – la sosirea unui caracter, acesta e preluat din registrul de date și stocat în bufferul de recepție.*

*Transmiterea de șiruri și caractere individuale trebuie să folosească funcții non-blocante (**USART_transmit_char()**, **USART_transmit_string()**), iar recepția trebuie să permită citirea ulterioară a datelor stocate în buffer (**USART_receive_char()**, **USART_receive_string()**) fără a opri rularea aplicației principale.*

***Conectări hardware:** Conectarea se face pe portul **microBUS 3**, unde pinii **TXD** și **RXD** ai microcontrolerului sunt legați la interfața de comunicație serială a plăcii. În cazul testării cu PC-ul, se poate folosi un cablu **USB-UART** pentru a deschide un terminal serial la **9600 baud**. Conectarea se poate face, la voia studentului, pe orice alt microBUS, cu condiția modificării pinilor aferenți microBUS-ului respectiv.*

`usart.h`

Header-ul **`usart.h`** declară funcțiile necesare pentru inițializarea și utilizarea modulului **USART** al microcontrolerului. Sunt incluse funcții pentru transmiterea și recepția de caractere individuale sau șiruri de caractere, precum și vectorii de întrerupere pentru evenimentele de tip **Data Register Empty** și **Receive Complete**. Constantele **`F_OSC`**, **`BAUD`** și **`BAUD_RATE`** permit configurarea ratei de baud în funcție de frecvența de tact a sistemului.

```
/*----------------------------------------------------------------------------
 * Fișier: usart.h
 * Utilizat pentru declararea funcțiilor USART
 *--------------------------------------------------------------------------*/

#ifndef __USART__
#define __USART__

/*----------------------------------------------------------------------------
 * Includes
 *--------------------------------------------------------------------------*/

#include <inavr.h>
#include <ioavr.h>
#include "roundBuff.h"

/*----------------------------------------------------------------------------
 * Public defines
 *--------------------------------------------------------------------------*/

#define F_OSC 16000000UL
#define BAUD 9600
#define BAUD_RATE (F_OSC/16/BAUD - 1)

/*-------------------------------------------------------------------------
 * Public (exported) functions
 *-----------------------------------------------------------------------*/

// Funcție folosită la inițializarea modulului USART
void USART_initialize(uint16_t baud_rate);
// Funcție folosită la ransmiterea unui singur caracter
uint16_t USART_transmit_char(uint8_t c);

// Funcție folosită la transmiterea a mai multor caractere
uint16_t USART_transmit_string(uint8_t *s, int16_t length);

// Funcție folosită pentru recepția unui singur caracter
uint16_t USART_receive_char(uint8_t *c);

// Funcție folosită pentru recepția a mai multor caractere
uint16_t USART_receive_string(uint8_t *c, uint16_t length);
```

```
// Funcție folosită la întreruperea pentru DATA REGISTER EMPTY
#pragma vector = USART1_TX_vect
__interrupt void USART1_TX_ISR(void);

// Funcție folosită la întreruperea pentru RECEIVE COMPLETE
#pragma vector = USART1_RX_vect
__interrupt void USART1_RX_ISR(void);

#endif
```

usart.c

Fișierul **usart.c** conține implementarea funcțiilor declarate în **usart.h**, asigurând funcționalitatea completă pentru comunicarea serială prin interfața USART1 a microcontrolerului. Sunt definite rutine pentru inițializarea modulului (**USART_initialize**), transmiterea și recepția de caractere individuale și șiruri de caractere, precum și întreruperile asociate transmisiei (**USART1_TX_vect**) și recepției (**USART1_RX_vect**). Transmiterea și recepția sunt gestionate asincron, folosind două buffer-e circulare (**tx_buffer, rx_buffer**), care asigură stocarea temporară a datelor și decuplarea logicii de comunicare de restul aplicației. Întreruperile sunt folosite pentru a trimite și a primi caractere fără blocare, iar logica include protecție prin dezactivarea și reactivarea întreruperilor în secțiunile critice.

```
/*-----------------------------------------------------------------------------
 * Fișier: usart.c
 * Utilizat pentru definirea funcțiilor USART din usart.h
 *---------------------------------------------------------------------------*/

/*-----------------------------------------------------------------------------
 * Includes
 *---------------------------------------------------------------------------*/

// General
#include "usart.h"

/*-----------------------------------------------------------------------------
 * (Private) defines
 *---------------------------------------------------------------------------*/

/*
 * Acest cod este configurat pentru portul microbus 1 de pe placă, folosind
 * controalele UART1 ale procesorului.
 */

#define DATA_REGISTER_EMPTY     (UCSR1A & (1 << UDRE1))
#define SET_TRANSMITTER             (1 << TXEN1)
#define SET_RECEIVER                (1 << RXEN1)
#define SET_RX_COMPLETE             (1 << RXCIE1)
#define SET_TX_COMPLETE             (1 << TXCIE1)
#define SET_BAUD_H(BAUDRATE) UBRR1H = (uint8_t)(BAUDRATE >> 8)
#define SET_BAUD_L(BAUDRATE) UBRR1L = (uint8_t)(BAUDRATE & 0xFF)

/*-----------------------------------------------------------------------------
 * Public variables
 *---------------------------------------------------------------------------*/

roundBuff_s tx_buffer;
roundBuff_s rx_buffer;

/*-----------------------------------------------------------------------------
 * Private functions
 *---------------------------------------------------------------------------*/
```

```
void usart_send_next()
{
  if (!isEmpty(&tx_buffer))
    {

        // Se pune caracterul în registrul de date pentru a fi trimis
        UDR1 = pop(&tx_buffer);
    }
}

/*--------------------------------------------------------------------------
 * Public functions
 *-------------------------------------------------------------------------*/

// Funcție folosită la inițializarea modulului USART
void USART_initialize(uint16_t baud_rate)
{
  // Se configurează baud rate-ul
   tx_buffer.head = 0;
   tx_buffer.tail = 0;
   rx_buffer.head = 0;
   rx_buffer.tail = 0;

   SET_BAUD_H(baud_rate);
   SET_BAUD_L(baud_rate);

   /*
    * Se pornește transmițătorul și receptorul, iar apoi se activează
    * întreruperea la recepție și la transmisie.
    */
 UCSR1B = SET_TRANSMITTER | SET_RECEIVER | SET_RX_COMPLETE | SET_TX_COMPLETE;
   // Se configurează paritatea pe par
    UCSR1C |= (1<<UPM11) | (1<<UCSZ11) | (1<<UCSZ10);
   // Se setează pinul TXD: ieșire
    DDRD |= (1 << PD3);
   // Se setează pinul RXD: intrare
    DDRD &= ~(1 << PD2);
}

// Funcție folosită la ransmiterea unui singur caracter
uint16_t USART_transmit_char(uint8_t c)
{
    // Se dezactivează întreruperile pentru acces sigur la buffer
    __disable_interrupt();
    uint16_t verif;

    // Se verifică dacă caracterul nu este terminator de șir
    if(c != '\0')
        // Se adaugă caracterul în bufferul de transmisie
        verif = push(&tx_buffer, c);

    // Dacă registrul de date este gol și caracterul a fost adăugat cu succes
    if(DATA_REGISTER_EMPTY && verif != 0)
        // Se trimite imediat caracterul următor din buffer
        usart_send_next();

    // Se reactivează întreruperile
    __enable_interrupt();
    // Se returnează starea operației (0 = eșuat, 1 = reușit)
    return verif;
}
```

```
// Funcție folosită la transmiterea a mai multor caractere
uint16_t USART_transmit_string(uint8_t *s, int16_t length)
{
    // Se dezactivează întreruperile pentru acces sigur la buffer
    __disable_interrupt();

    // Se adaugă șirul de caractere în bufferul de transmisie
    uint16_t verif = push_vec(&tx_buffer, s, length);

    // Dacă registrul de date e gol și caracterele au fost adăugate cu succes
    if(DATA_REGISTER_EMPTY && verif != 0)
        // Se trimite imediat primul caracter din buffer
        usart_send_next();

    // Se reactivează întreruperile
    __enable_interrupt();

    // Se returnează numărul de caractere adăugate cu succes în buffer
    return verif;
}

// Funcție folosită pentru recepția a mai multor caractere
uint16_t USART_receive_string(uint8_t *c, uint16_t length)
{
    // Se dezactivează întreruperile pentru acces sigur la buffer
    __disable_interrupt();

    uint16_t i;

    // Se parcurge până la 'length' caractere
    for(i = 0; i < length; i++)
    {
        // Se scoate un caracter din bufferul de recepție
        uint8_t chr = pop(&rx_buffer);
        // Dacă bufferul e gol sau s-a ajuns la '\0', se oprește citirea
        if(chr == 0)
            break;
        // Se salvează caracterul citit în șirul furnizat de utilizator
        c[i] = chr;
    }
    // Se reactivează întreruperile
    __enable_interrupt();

    // Se returnează numărul de caractere citite
    return i;
}

/*
 * Întrerupere pentru "Data Register Empty" (UDRIE), executată când
 * transmițătorul este gata să trimită un nou caracter.
 */
#pragma vector = USART1_TX_vect
__interrupt void USART1_TX_ISR(void)
{
    // Se trimite următorul caracter din bufferul de transmisie
    usart_send_next();
}
```

```
/*
 * Întrerupere pentru "Receive Complete" (RXC), executată atunci când a sosit
 * un caracter în registrul de recepție. Se preia caracterul din UDR1 și se
 * stochează în bufferul de recepție.
 */
#pragma vector = USART1_RX_vect
__interrupt void USART1_RX_ISR(void)
{
    // Se citește caracterul primit
    uint8_t received_char = UDR1;
    // Se adaugă caracterul în bufferul de recepție
    push(&rx_buffer, received_char);
}
```

`round_buff.h`

Fișierul **`round_buff.h`** definește structura și interfața pentru implementarea bufferelor circulare utilizate în comunicația serială. Acesta include definiții pentru dimensiunea bufferului, structura bufferului circular (**head**, **tail** și **array-ul de date**), tipurile de date publice și prototipurile funcțiilor pentru adăugarea (**`push()`**), extragerea (**`pop()`**) și verificarea (**`is_empty()`** / **`is_full()`**) stării bufferului. De asemenea, declară bufferele globale pentru transmisie și recepție, care sunt folosite de modulul **USART**.

```
/*---------------------------------------------------------------------------
 * Fișier: round_buff.h
 * Utilizat pentru declararea funcțiilor și structurii bufferului circular
 *-------------------------------------------------------------------------*/

#ifndef __ROUND_BUFF__
#define __ROUND_BUFF__

/*---------------------------------------------------------------------------
 * Includes
 *-------------------------------------------------------------------------*/

// Biblioteca standard
#include <stdint.h>
#include <stddef.h>
#include <inavr.h>
#include <ioavr.h>

/*---------------------------------------------------------------------------
 * Public defines
 *-------------------------------------------------------------------------*/

// Dimensiunea bufferului circular
#define BUFFER_SIZE 16

/*---------------------------------------------------------------------------
 * Data structures
 *-------------------------------------------------------------------------*/

// Structură utilizată pentru buffer circular
struct round_buff {
    uint8_t buffer[BUFFER_SIZE]; // Array-ul de date
    uint8_t head;                // Indexul de citire
    uint8_t tail;                // Indexul de scriere
};

/*---------------------------------------------------------------------------
 * Type definitions
 *-------------------------------------------------------------------------*/
```

```
// Tip definit pentru buffer circular
typedef struct round_buff round_buff_s;

/*--------------------------------------------------------------------------
 * Public (exported) variables
 *------------------------------------------------------------------------*/

// Buffer folosit pentru transmisie
extern round_buff_s tx_buffer;

// Buffer folosit pentru recepție
extern round_buff_s rx_buffer;
/*--------------------------------------------------------------------------
 * Public (exported) functions
 *------------------------------------------------------------------------*/

// Funcție folosită pentru adăugarea unui caracter în buffer
int16_t push(round_buff_s *in_buffer, uint8_t data);

// Funcție folosită pentru extragerea unui caracter din buffer
uint8_t pop(round_buff_s *in_buffer);

// Funcție folosită pentru adăugarea unui șir de caractere în buffer
int16_t push_vec(round_buff_s *in_buffer, uint8_t data[], int16_t length);

// Funcție folosită pentru verificarea dacă bufferul este gol
int16_t is_empty(round_buff_s *in_buffer);

// Funcție folosită pentru verificarea dacă bufferul este plin
int16_t is_full(round_buff_s *in_buffer);

#endif
```

round_buff.c

Fișierul **round_buff.c** conține implementarea funcțiilor pentru gestionarea bufferelor circulare utilizate în comunicația serială. Sunt definite funcțiile pentru adăugarea unui caracter (**push()**), extragerea unui caracter (**pop()**), adăugarea unui șir de caractere (**push_vec()**) și verificarea stării bufferului (**is_empty()** și **is_full()**). Funcțiile asigură manipularea circulară a indicilor **head** și **tail** și permit stocarea temporară a datelor fără blocarea execuției principale.

```
/*--------------------------------------------------------------------------
 * Fișier: round_buff.c
 * Utilizat pentru definirea funcțiilor și structurilor din round_buff.h
 *------------------------------------------------------------------------*/

/*--------------------------------------------------------------------------
 * Includes
 *------------------------------------------------------------------------*/

// Include fișierul de header pentru buffer circular
#include "roundBuff.h"

/*--------------------------------------------------------------------------
 * Public functions
 *------------------------------------------------------------------------*/
```

```
// Funcție folosită pentru adăugarea unui caracter în bufferul circular
int16_t push(round_buff_s *inBuffer, uint8_t data)
{
    /*
     * Se calculează poziția următoare a indicelui head folosind operația
     * modulo pentru comportament circular.
     */
    uint8_t next_head = (inBuffer->head + 1) % BUFFER_SIZE;

    /*
     * Dacă bufferul nu este plin, se inserează caracterul și
     * se actualizează indicele head.
     */
    if(!is_full(inBuffer))
    {
        inBuffer->buffer[inBuffer->head] = data;
        inBuffer->head = next_head;

        // Returnează 1 pentru succes
        return 1;
    }
    else
        // Returnează 0 dacă bufferul este plin
        return 0;
}

// Funcție folosită pentru extragerea unui caracter din buffer
uint8_t pop(round_buff_s *inBuffer)
{
    /*
     * Dacă bufferul nu este gol, se preia caracterul din poziția tail
     * și se actualizează indicele tail pentru comportament circular.
     */
    if(!is_empty(inBuffer))
    {
        uint8_t data = inBuffer->buffer[inBuffer->tail];
        inBuffer->tail = (inBuffer->tail + 1) % BUFFER_SIZE;
        // Returnează caracterul extras
        return data;
    }
    else
    {
        // Returnează 0 dacă bufferul este gol
        return 0;
    }
}

// Funcție folosită pentru adăugarea unui șir de caractere în buffer
int16_t push_vec(round_buff_s *inBuffer, uint8_t data[], int16_t length)
{
    int16_t i;
    /*
     * Se parcurge fiecare caracter din șir și se încearcă adăugarea lui.
     * În caz că bufferul se umple, se oprește inserarea.
     */
    for(i = 0; i < length; i++)
    {
        uint8_t verif = push(inBuffer, data[i]);
        if(verif == 0)
            break;
    }
```

```
    // Returnează numărul de caractere adăugate cu succes
    return i;
}

// Funcție folosită pentru verificarea dacă bufferul este gol
int16_t is_empty(round_buff_s *inBuffer)
{
    return (inBuffer->head == inBuffer->tail);
}

// Funcție folosită pentru verificarea dacă bufferul este plin
int16_t is_full(round_buff_s *inBuffer)
{
    return ((inBuffer->head + 1) % BUFFER_SIZE) == inBuffer->tail;
}
```

mylib.h

Header-ul **mylib.h** declară funcția **my_print()**, utilizată pentru afișarea variabilelor prin interfața **USART**, în funcție de tipul acestora. Tipul este specificat printr-un enum dedicat (**Tipuri**), care permite identificarea valorii ca întreg (**int**), hexazecimal (**hex**), număr real (**double**) sau caracter (**char**), oferind o interfață abstractă pentru funcționalitatea de printare.

```
/*----------------------------------------------------------------------------
 * Fișier: mylib.h
 * Utilizat pentru declararea funcției my_print
 *--------------------------------------------------------------------------*/

#ifndef __MYLIB__
#define __MYLIB__

/*----------------------------------------------------------------------------
 * Includes
 *--------------------------------------------------------------------------*/

// Compiler
#include <stdint.h>
#include <inavr.h>
#include <ioavr.h>

// General
#include "usart.h"

/*----------------------------------------------------------------------------
 * Data structures
 *--------------------------------------------------------------------------*/
typedef enum Tip {
  INTEGER,
  HEX,
  DOUBLE,
  CHAR
} Tipuri;

/*----------------------------------------------------------------------------
 * Public (exported) functions
 *--------------------------------------------------------------------------*/
```

```
/*
 * Funcția de print care folosește USART
 * Tip definește tipul variabilei: 0 = int, 1 = hexa, 2 = double, 3 = char
 */
void my_print(Tipuri tip, void *val);

#endif
```

mylib.c

Fișierul **mylib.c** conține implementarea funcției **my_print()** declarate în **mylib.h**, oferind suport pentru transmiterea de date seriale prin interfața USART. În funcție de tipul specificat (**INTEGER**, **HEX**, **DOUBLE** sau **CHAR**), fișierul utilizează funcții specializate pentru a converti și transmite valori numerice întregi, hexazecimale, reale (cu două zecimale) sau caractere.

```
/*-----------------------------------------------------------------------------
 * Fișier: mylib.c
 * Utilizat pentru definirea funcției my_print
 *---------------------------------------------------------------------------*/

/*-----------------------------------------------------------------------------
 * Includes
 *---------------------------------------------------------------------------*/

// General
#include "mylib.h"

/*-----------------------------------------------------------------------------
 * Private (static) variables
 *---------------------------------------------------------------------------*/

// Buffer auxiliar pentru construirea șirului de caractere
static uint8_t aux1[10] = {0}, aux2[10] = {0};

/*-----------------------------------------------------------------------------
 * Private functions
 *---------------------------------------------------------------------------*/

// Funcția de transmitere a unui număr întreg pe serială
void integerTransmit(void *p)
{
    // Se inițializează indicii pentru construcția șirului de caractere
    int16_t index = 0, i = 0;

    // Se preia valoarea întreagă din pointer
    int16_t x = (*(int16_t *)(p));

    // Se tratează cazul numerelor negative
    if(x < 0)
    {
        aux1[index] = '-';  // Se adaugă semnul minus
        index++;
        x *= (-1);          // Se convertește la valoare pozitivă
    }

    // Se extrag cifrele numărului în ordine inversă
    do
    {
        // Se convertește cifra la caracter ASCII
        uint8_t c = x % 10 + '0';
```

```
        // Se stochează temporar în buffer auxiliar
        aux2[i] = c;
        i++;

        // Se reduce numărul
        x = x / 10;
    } while(x != 0);


    // Se copiază cifrele în ordinea corectă în bufferul final
    for(int16_t j = i - 1; j >= 0; j--)
    {
        aux1[index] = aux2[j];
        index++;
    }

    // Se adaugă caracterele de sfârșit de linie
    aux1[index] = '\n';
    index++;
    aux1[index] = '\r';
    index++;

    // Se transmite șirul format către modulul USART
    USART_transmit_string(aux1, index);
}

// Funcția de transmitere a unui număr hexazecimal pe serială
void hexadecimalTransmit(void *p)
{
    // Se preia valoarea întreagă din pointer
    int16_t x = *((int16_t *)(p));
    int16_t index = 0, i = 0;

    // Se adaugă prefixul "0x" pentru format hexazecimal
    aux1[index] = '0';
    index++;
    aux1[index] = 'x';
    index++;

    // Se extrag cifrele hexazecimale în ordine inversă
    do
    {
        uint8_t a = x & 0x0F;  // Se preiau cei 4 biți cei mai mici

        // Se convertește valoarea la caracter ASCII
        if(a <= 9)
        {
            aux2[i] = a + '0';
        }
        else
        {
            aux2[i] = a + 'A' - 10;
        }
        i++;
        x >>= 4;  // Se face shiftare pentru următoarea cifră
    } while(x != 0);

    // Se copiază cifrele în ordinea corectă
    for(int16_t j = i - 1; j >= 0; j--)
    {
        aux1[index] = aux2[j];
```

```
        index++;
    }
    // Se transmite șirul format către modulul USART
    USART_transmit_string(aux1, index);
}

// Funcția de transmitere a unui număr de tip double pe serială
void doubleTransmit(void *p)
{
    // Se preia valoarea double din pointer
    double x = (*(double *)(p));
    int16_t index = 0;
    int16_t i = 0;

    // Se tratează cazul numerelor negative
    if (x < 0)
    {
        aux1[index++] = '-';
        x = -x;
    }

    // Se separă partea întreagă de partea fracționară
    int16_t int_part = (int16_t) x;
    double frac = x - int_part;
    // Se păstrează două zecimale
    int16_t frac_part = (int16_t)(frac * 100 + 0.5);

    // Se procesează partea întreagă
    if (int_part == 0)
    {
        aux1[index++] = '0';
    }
    else
    {
        while (int_part != 0)
        {
            // Se convertește cifra la caracter ASCII
            uint8_t c = int_part % 10 + '0';
            // Se stochează temporar în buffer auxiliar
            aux2[i++] = c;
            // Se reduce valoarea
            int_part /= 10;
        }
        // Se copiază cifrele în ordinea corectă
        for (int16_t j = i - 1; j >= 0; j--)
        {
            aux1[index++] = aux2[j];
        }
    }

    // Se adaugă separatorul zecimal
    aux1[index++] = '.';

    i = 0;

    // Se procesează partea fracționară
    if (frac_part == 0)
    {
        aux1[index++] = '0';
        aux1[index++] = '0';
    }
```

```
    else
    {
        if (frac_part < 10)
        {
            aux1[index++] = '0';  // Se adaugă zeroul pentru o cifră
        }
        // Se extrag cifrele fracționare în ordine inversă
        while (frac_part != 0)
        {
            uint8_t c = frac_part % 10 + '0';
            aux2[i++] = c;
            frac_part /= 10;
        }

        // Se copiază cifrele în ordinea corectă
        for (int16_t j = i - 1; j >= 0; j--)
        {
            aux1[index++] = aux2[j];
        }
    }

    // Se adaugă caracterele de sfârșit de linie
    aux1[index++] = '\n';
    aux1[index++] = '\r';

    // Se transmite șirul format către modulul USART
    USART_transmit_string(aux1, index);
}

// Se transmite un caracter pe serială
void characterTransmit(void *p)
{
    // Se preia caracterul din pointer
    int8_t x = (*(int8_t *)(p));
    int16_t index = 0;

    // Se adaugă caracterul în buffer
    aux1[index++] = x;

    // Se transmite caracterul către modulul USART
    USART_transmit_string(aux1, index);
}

/*----------------------------------------------------------------------------
 * (Public) Functions (the ones from .h)
 *---------------------------------------------------------------------------*/

// Funcție de tip wrapper prin care se transmit valori de diverse tipuri
void myprint(Tipuri tip, void *val)
{
    switch(tip)
    {
        case INTEGER:
            // Se transmite un număr întreg
            integerTransmit(val);
            break;
        case HEX:
            // Se transmite un număr hexazecimal
            hexadecimalTransmit(val);
            break;
```

```
        case DOUBLE:
            // Se transmite un număr double
            doubleTransmit(val);
            break;


        case CHAR:
            // Se transmite un caracter
            characterTransmit(val);
            break;
    }
}
```

main.c

main.c este punctul de intrare al aplicației, unde se inițializează interfața **USART** și se activează întreruperile globale. La pornire, se definește o variabilă numerică ce crește treptat cu un pas fix și este transmisă periodic prin funcția **myprint()**. La atingerea limitelor impuse (**0.0** și **50.0**), direcția de incrementare se inversează, obținându-se un ciclu oscilant. Între transmisiile succesive se introduce o întârziere controlată, ceea ce reglează ritmul de afișare pe interfața serială.

```
/*----------------------------------------------------------------------------
 * Fișier: main.c
 * Utilizat pentru pornirea aplicației de scriere serială
 *------------------------------------------------------------------------*/

#include "mylib.h"
#include "usart.h"

int16_t main(void)
{
  // Se inițializează USART-ul cu baud rate-ul definit
  USART_initialize(BAUD_RATE);

  // Se activează întreruperile globale
  __enable_interrupt();

  // Se definește variabila transmisă
  double a = 0.0;
  // Se definește pasul de incrementare/decrementare
  double pas = 0.5;

  while(1)
  {
      // Se transmite valoarea variabilei prin USART
      myprint(DOUBLE, &a);

      // Se actualizează valoarea variabilei
      a += pas;

      // Dacă se ating limitele, direcția de variație se inversează
      if(a >= 50.0 || a <= 0.0)
        pas *= -1;

      // Se introduce o întârziere pentru stabilirea ritmului de transmitere
      __delay_cycles(1600000);
  }
}
```

5.15 MĂSURAREA CU OSCILOSCOPUL

În continuare este descris felul în care se poate măsura transmiterea serială **USART**, la voia studentului, folosind **osciloscopul**.

O sondă va fi conectată de pe canalul 1 (**CH1**) al osciloscopului la pinul de **Tx** de pe **UNI Clicker**, iar a doua sondă va fi conectată de pe canalul 2 (**CH2**) al osciloscopului la pinul **Rx** de pe **UNI Clicker**.

Intrăm din meniu **Analysis → Decode...**

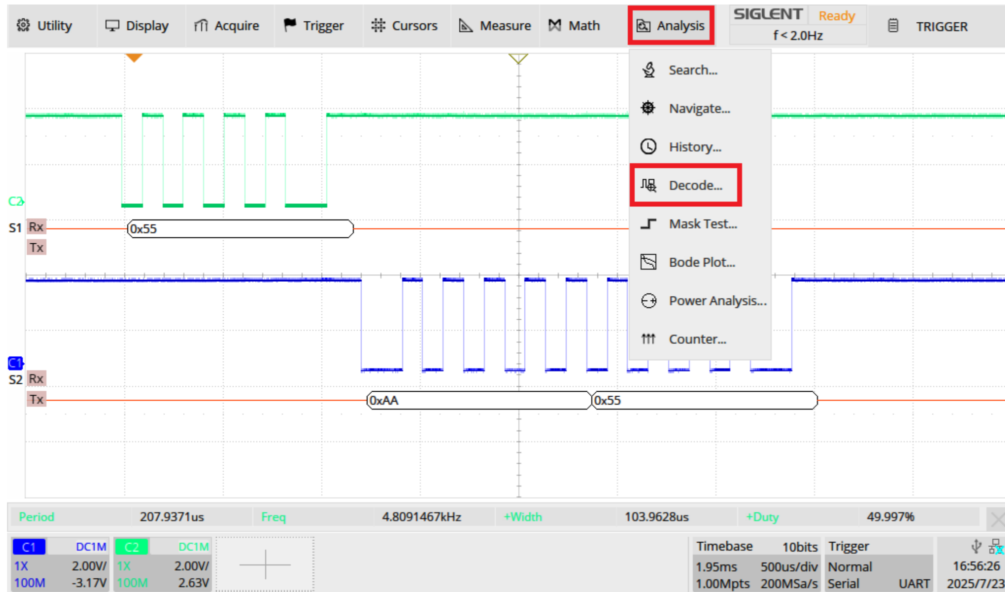


Figura 5.20 – Meniul Analysis

După care: **Bus → Bus1**, **Bus Operation → On** și **Bus Protocol → UART**.

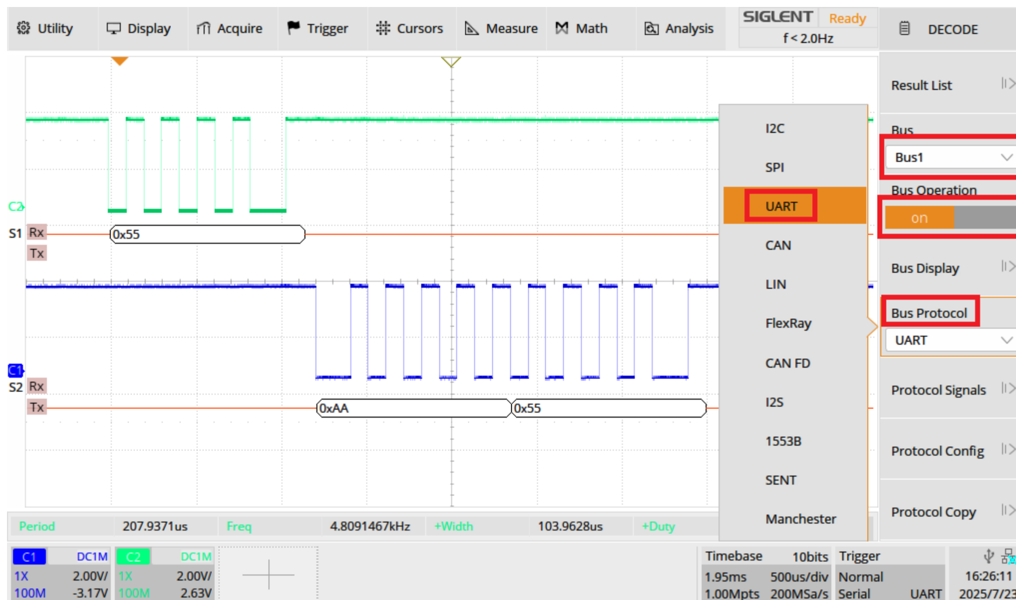


Figura 5.21 – Setări UART Bus 1

Apoi, **Protocol Signals → RX → C2 → Threshold → 2.00 V**.

Figura 5.22 – Meniul Protocol Signals

Protocol Config → Baud → valoarea configurată în cod (în acest caz 9600 bits/s)
→ Data Length → valoarea configurată în cod (în acest caz 8)
→ Parity Check → valoarea configurată în cod (în acest caz Even)
→ Stop bit → valoarea configurată în cod (în acest caz 1)
→ Idle Level → High
→ Bit Order → LSB

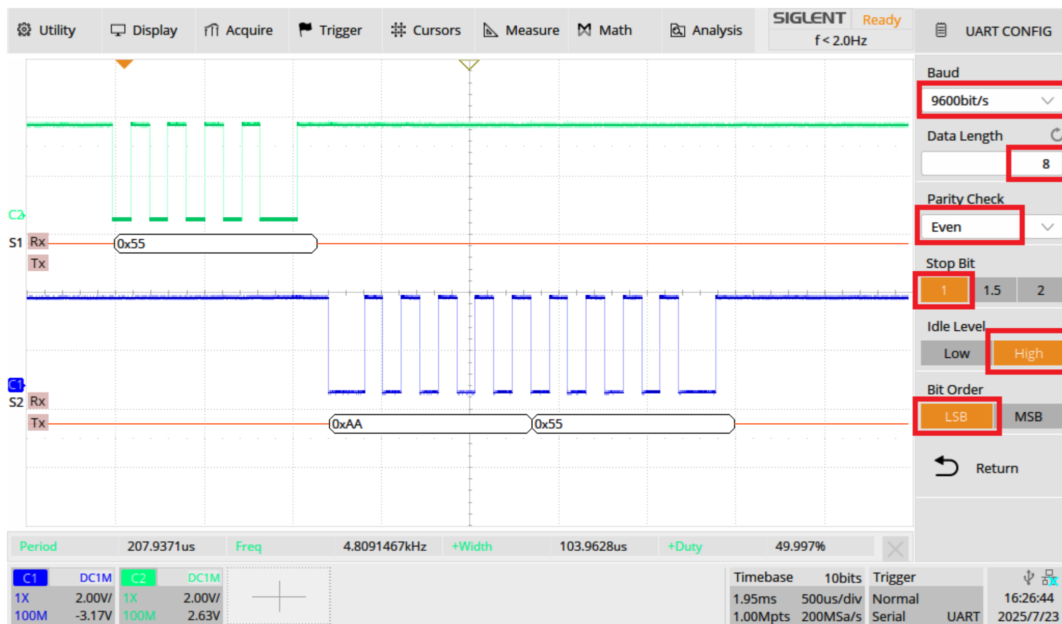


Figura 5.23 – Meniul Protocol Config

Bus → Bus2
Bus Operation → On

Figura 5.24 – Setări UART Bus 2

Protocol Signals → TX → C1
→ Threshold → 2.00 V

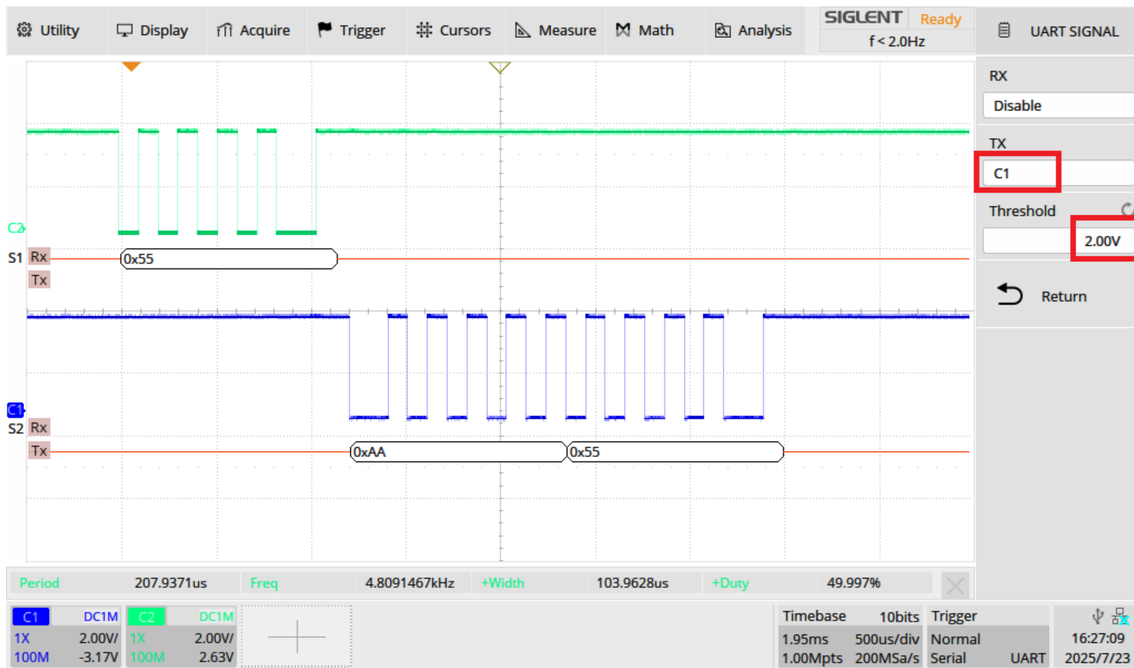


Figura 5.25 – Meniul Protocol Signals

Intrăm din meniu → **Trigger → Menu…**

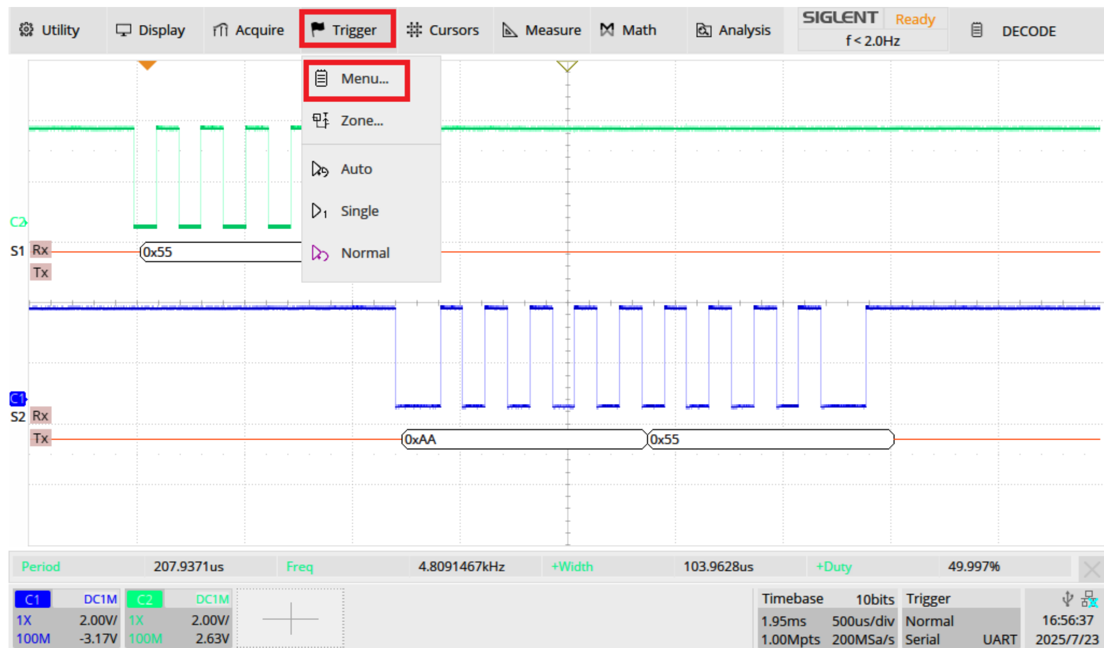


Figura 5.26 – Meniul Trigger

Trigger → Type → Serial
→ Protocol → UART

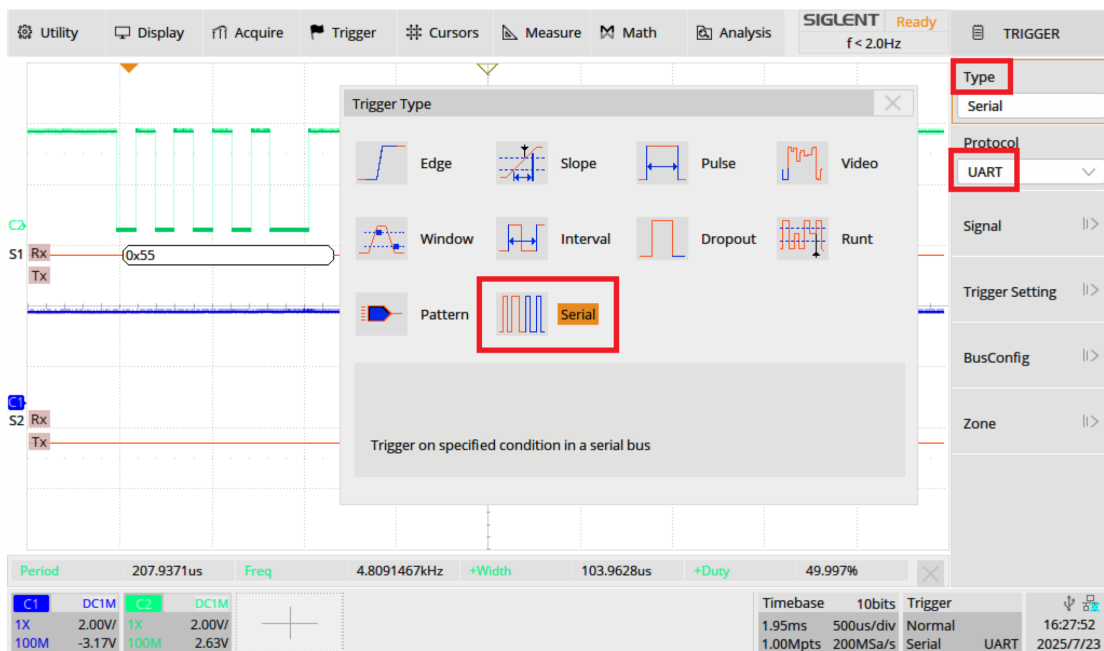


Figura 5.27 – Setări Trigger

Trigger → Trigger Settings → Source Type → RX
→ Condition → Start

Figura 5.28 – Setări Condition

5.16 BIBLIOGRAFIE

1. "8-bit AVR Microcontroller ATmega 1280 Datasheet", Microchip Technology
2. "Schematic for UNI Clicker", Mikroe
3. "Schematic for ATmega1280: SiBrain", Mikroe
4. "Universal Asynchronous Receiver Transmitter", Wikipedia
5. "Introduction to SPI Interface", Analog Devices
6. "UART Communication", Circuit Basics

6. TIMER / COUNTER

6.1 UNDE SE FOLOSEȘTE TIMER / COUNTER?

Timer / Counter-ul este utilizat în multe aplicații electronice şi sisteme integrate, fiind responsabil pentru măsurarea precisă a timpului sau numărarea evenimentelor externe:

- **Ceasuri și cronometre digitale:** pentru a măsura timpul scurs cu precizie;
- **Generarea semnalelor PWM:** controlul motoarelor sau al intensității luminii LED-urilor;
- **Evenimente externe:** numărarea impulsurilor de la senzori (ex. encoder de rotație, viteză, debitmetru);
- **Comunicații:** temporizare exactă pentru protocoale (baud rate, intervale de sincronizare).

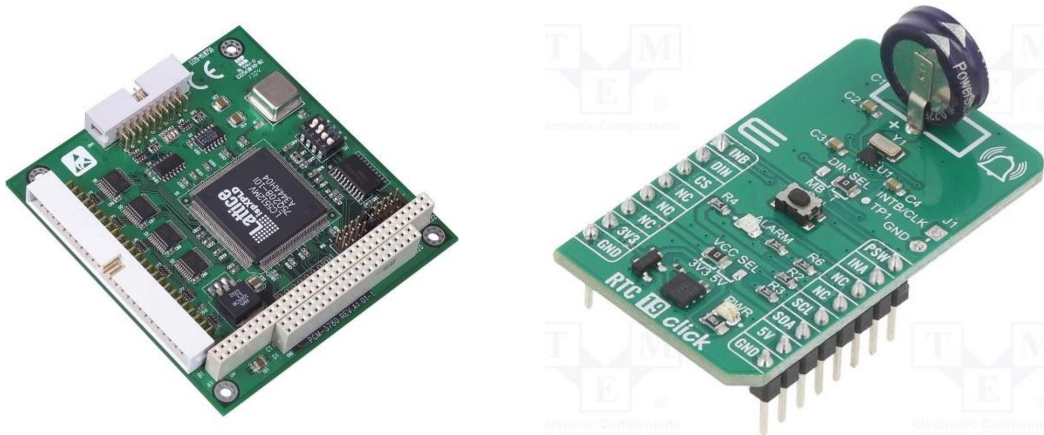


Figura 6.1 – Un modul industrial PC/104 (stânga) și un RTC Click (dreapta), dedicate funcțiilor de Timer

6.2 CUNOȘTINȚE NECESARE

Pentru a putea parcurge acest capitol, este necesar să cunoașteți următoarele subiecte din capitolele anterioare, și nu numai:

- Utilizarea întreruperilor;
- Utilizarea osciloscopului pentru măsurarea semnalelor PWM;
- Utilizarea kit-ului hardware Mikroe.

6.3 ABSTRACT

Acest capitol descrie utilizarea pe platforma **ATmega1280** a tehnicii **Pulse Width Modulation (PWM)**, o metodă eficientă de a controla puterea aplicată unui dispozitiv electric prin varierea duratei semnalului digital în stare activă. Se va explora cum se configurează și utilizează **PWM** folosind timerele hardware integrate ale microcontrolerului pentru a genera semnale cu frecvență constantă și duty cycle variabil, iar pentru o exemplificare amănunțită, s-a propus:

- generarea unui semnal PWM software;
- controlarea intensității unui LED conectat;
- realizarea unui periodmetru;
- generarea unui semnal modulat.

6.4 HARDWARE ȘI SOFTWARE NECESAR

- ATmega1280 SiBRAIN;
- UNI Clicker;
- Atmel ICE;
- IAR *Embedded Workbench* IDE 7.30.5;
- Osciloscop.

6.5 NOȚIUNI INTRODUCTIVE

Definiții

Frecvența este numărul de apariții ale unui eveniment repetitiv pe unitatea de timp. Perioada este timpul necesar pentru a completa un ciclu al unei oscilații sau rotații. Relația dintre frecvență și perioadă este:

$$\mathbf{T} = \frac{1}{\text{frecvență}}$$

Factorul de umplere (duty cycle) este valoarea care indică cât din întreaga perioadă semnalul are valoarea "1" și se exprimă în procente.

$$\mathbf{T} = \frac{p}{p+q}\ x\ 100\%, \text{ unde:}$$

- **p este timpul de "1" (timpul în care ne aflăm pe nivelul 1 logic);**
- **q este timpul de "0" (timpul în care ne aflăm pe nivelul 0 logic);**

Notă: T = p + q.

PWM (Pulse Width Modulation) este o tehnică esențială în controlul dispozitivelor electronice care permite varierea eficientă a tensiunii medii aplicate unei sarcini.

Aceasta este realizată prin alternarea rapidă între starea **HIGH** și **LOW** a unui semnal digital. Durata în care semnalul este **HIGH** (**duty cycle**) determină puterea medie aplicată. **PWM** acolo unde este necesar un control precis al puterii fără a risipi energie. În implementarea hardware se ține cont, în aproape toate microcontrolerele, de registrele **Timer / Counter**. Astfel, generarea unei frecvențe prin **PWM** ține cont de frecvența microcontrolerului și capacitățile registrelor **Timer / Counter**: **rezoluție**, posibilitate de **output compare**, posibilitate de **modificare a duty cycle-ului**.

Pentru a calcula **perioada**, **frecvența** și **factorul de umplere** al unui semnal **PWM** generat de un microprocesor **ATmega1280** folosind un timer de **8 sau 16 biți**, se vor folosi următoarele formule, ținând cont de frecvența microprocesorului (**8 MHz în mod normal**, **16 MHz frecvența maximă**) și de **factorul de prescaler** care poate fi **selectat de utilizator**.

6.5.1 FRECVENȚA (HZ)

Frecvența unui semnal **PWM** depinde de valoarea **contorului / timer-ului**, de **prescaler** și de **frecvența clock-ului** sistemului. Formula generală este:

$$f_{PWM} = \frac{\text{frecvența microcontrolerului}}{\text{prescaler * valoarea maximă a timer-ului}} \quad (1)$$

De exemplu, pentru **ATmega1280** care funcționează la o frecvență la **8 MHz**, valoarea frecvenței microcontrolerului (sau frecvența ceasului – f_{CLK}) este:

$$f_{CLK} = 8\ \text{MHz} = 8 \times 10^6\ \text{Hz}$$

Pentru un timer pe **8 biți**:

$$f_{PWM} = \frac{f_{CLK}}{\text{prescaler} * 256} \quad (2)$$

Pentru un timer pe **16 biți**:

$$f_{PWM} = \frac{f_{CLK}}{\text{prescaler} * 65536} \quad (3)$$

6.5.2 PERIOADA (MILISECUNDE)

Perioada unui semnal **PWM** poate fi calculată astfel:

$$\textbf{Perioada } (T_{PWM}) = \frac{1}{f_{PWM}} \quad (4)$$

Notă: Atenție la unitățile de măsură! În cazul anterior, perioada este exprimată în **secunde**. Dacă se dorește obținerea perioadei în **milisecunde** (**ms**), multiplicăm cu **1000**:

$$\textbf{Perioada } (T_{PWM}) = \frac{1000}{f_{PWM}} \quad (5)$$

6.5.3 FACTORUL DE UMPLERE (DUTY CYCLE)

Definiție

Factorul de umplere (Duty Cycle) este procentul de timp în care semnalul PWM este "activ" (adică are valoare logică 1).

Dacă numărul de incrementări ale contorului **timer-ului** până când semnalul **PWM** se dezactivează este **OCRxn (Output Compare Register)**, atunci factorul de umplere se calculează astfel:

$$\textbf{Duty Cycle} = \left(\frac{\text{OCRxn}}{\text{valoarea maximă a timer-ului}}\right) \text{ x } 100\% \quad (6)$$

Pentru un timer de **8 biți**:

$$\textbf{Duty Cycle} = \left(\frac{\text{OCRxn}}{255}\right) \text{ x } 100\% \quad (7)$$

Pentru un timer de **16 biți**:

$$\textbf{Duty Cycle} = \left(\frac{\text{OCRxn}}{65535}\right) \text{ x } 100\% \quad (8)$$

6.6 MODURI DE OPERARE PWM

Microcontrolerul **ATmega1280** suportă generarea semnalelor **PWM** prin intermediul mai multor timere integrate. Acesta oferă mai multe moduri principale de generare a semnalelor PWM. Valorile counter-ului sunt clasificate astfel:

- **BOTTOM***:* reprezintă valoarea **minimă** a numărătorului (**counter**);
- **MAX***:* reprezintă valoarea **maximă** a numărătorului (**counter**);
- **TOP***:* counter-ul ajunge la valoarea **TOP** când devine **egal cu cea mai mare valoare din secvența de numărat**, iar această valoare poate fi **fixă** (**MAX**) sau **valoarea stocată** în registrul **OCRnx** în funcție de modul de operare.

6.6.1 CLEAR TIMER ON COMPARE MATCH (CTC)

În modul de operare **CTC** numărătorul crește de la **0**, dar în loc să ajungă la valoarea **MAX**, se resetează automat la **0** atunci când valoarea sa devine **egală** cu cea din registrul **OCRnx** (sau **ICRn** în cazul **TCNT** pe **16 biți**). Astfel, **OCRnx** acționează ca o valoare maximă personalizată (**TOP**).

Frecvența semnalului generat în modul **CTC** este determinată de valoarea **OCR** și de **prescaler-ul** utilizat pentru timer, fiind calculată cu formula:

$$F_{CTC} = \frac{F_{CLK}}{2 \text{ x N x } (\text{OCRnx} + 1)} \quad (9)$$, unde:

- **F_{CTC}** este frecvența semnalului generat în modul **CTC**;
- **F_{CLK}** este frecvența de lucru a microcontrolerului;
- **N** este valoarea factorului de scalare;
- **OCRxn** este valoarea din registrul de comparație.

Astfel, modul **CTC** este ideal pentru aplicații de generare a unor semnale **precise** și **periodice**, fără necesitatea unui control fin asupra **duty cycle-ului**.

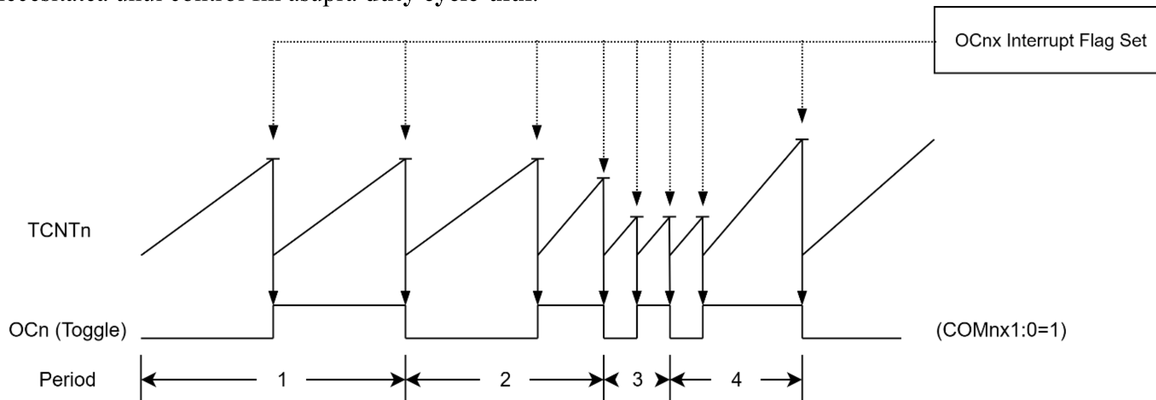


Figura 6.2 – Diagrama de timp pentru modul CTC

6.6.2 FAST PWM

Modul **Fast PWM** se bazează pe o numărare cu o singură pantă (**single-slope**): numărătorul crește de la **0** la valoarea **TOP**, apoi este resetat brusc înapoi la **0**, formând o undă de tip **"fierăstrău"**. Ieșirea **PWM** își schimbă starea la **începutul** ciclului (când numărătorul este la **0**) și la **potrivirea** dintre **TCNTn** și **OCRnx**.

Principalele caracteristici ale modului **Fast PWM** sunt:

1. Factorul de umplere a semnalului este controlat prin valoarea **OCRxn**. O valoare **OCRxn** mai **mică** înseamnă un factor de umplere mai **mic**, iar o valoare mai **mare OCRxn** înseamnă un factor de umplere mai **mare**.
2. Frecvența semnalului este determinată de valoarea **maximă** a contorului și de **factorul de scalare** a timer-ului. Spre deosebire de modul **CTC**, frecvența rămâne **constantă** în timpul funcționării, iar singura variabilă este factorul de umplere.

Acest mod este utilizat în aplicații unde sunt necesare semnale de înaltă frecvență și unde modificarea rapidă a factorului de umplere este **esențială**, cum ar fi în controlul motoarelor de curent continuu (**PWM controlat de microcontroler**), generarea de tonuri audio sau aplicațiile de iluminat controlate electronic.

Frecvența semnalului PWM în modul Fast PWM se calculează astfel:

$$F_{FastPWM} = \frac{F_{CLK}}{N \times (TOP + 1)} \quad \textbf{(10)}$$, unde:

- **$F_{FastPWM}$** este frecvența semnalului generat;
- **F_{CLK}** este frecvența de lucru a microcontrolerului;
- **N** este valoarea factorului de scalare;
- **TOP** este valoarea maximă la care ajunge contorul înainte de a fi resetat.

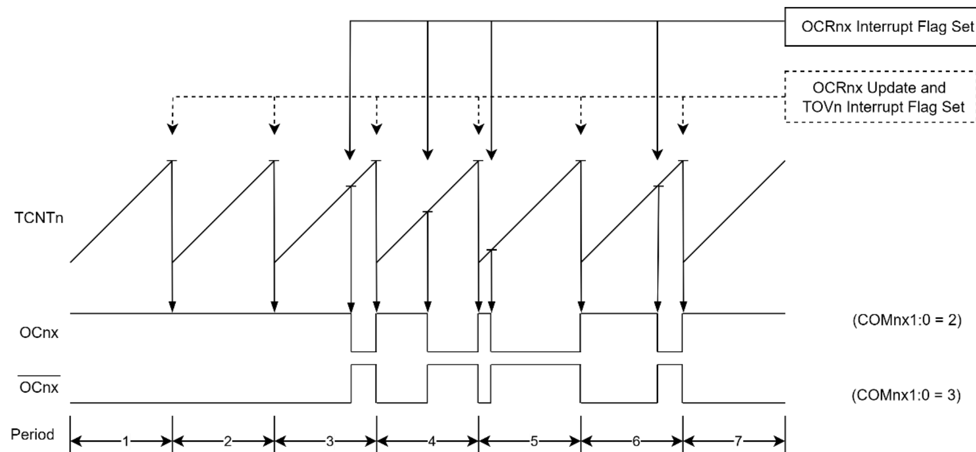


Figura 6.3 – Diagrama de timp pentru modul Fast PWM

6.6.3 PHASE CORRECT PWM

Acest mod de operare generează un semnal PWM **simetric**. **Phase Correct PWM** funcționează pe principiul numărării cu **pantă dublă (dual-slope)**: numărătorul crește de la **0** (**BOTTOM**) la **TOP**, apoi scade **simetric** înapoi la **0**, formând o undă **triunghiulară**. Ieșirea **PWM** își schimbă starea o dată pe panta **crescătoare** și a doua oară pe panta **descrescătoare**, la potrivirea cu **OCRnx**.

Acest mod este caracterizat de:

- Factorul de umplere este determinat de valoarea registrului **OCRnx**, la fel ca în modul **Fast PWM**, dar faza semnalului este ajustată pentru a minimiza **distorsiunea** și **zgomotul de comutație**. Aceasta asigură o **simetrie perfectă** în distribuția timpului între stările **HIGH** și **LOW**.
- Frecvența semnalului este de obicei mai **mică** decât în modul **Fast PWM**, deoarece contorul efectuează atât o contorizare **ascendentă**, cât și una **descendentă** într-un ciclu complet.

Modul **Phase Correct PWM** este ideal pentru aplicații care necesită o comutație mai lină a semnalului și minimizarea variațiilor rapide de tensiune, cum ar fi în controlul motoarelor, unde este importantă reducerea zgomotului și a interferențelor electromagnetice (**EMI**).

Frecvența semnalului **PWM** în modul **Phase Correct** este calculată astfel:

$$F_{PhaseCorrect} = \frac{F_{CLK}}{2x\ N\ x\ (TOP+1)} \quad \textbf{(11)}$$, unde:

- $F_{PhaseCorrect}$ este frecvența semnalului generat,
- F_{CLK} este frecvența de lucru a microcontrolerului,
- **N** este valoarea factorului de scalare,
- **TOP** este valoarea maximă la care contorul oscilă înainte de a schimba direcția.

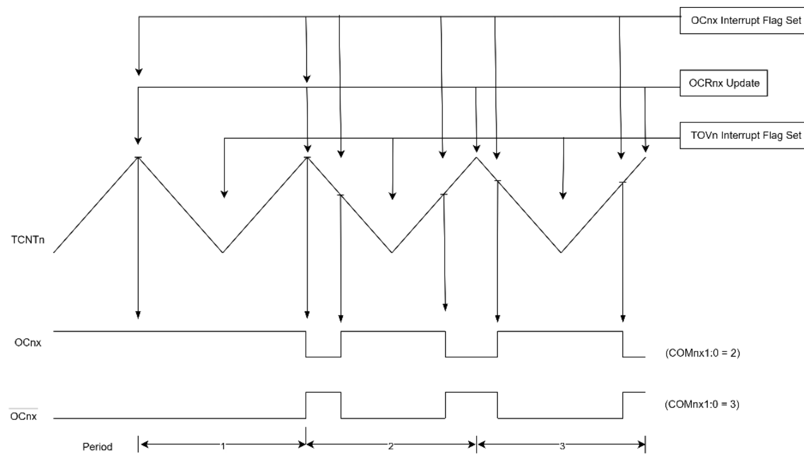


Figura 6.4 – Diagrama de timp pentru modul Phase Correct PWM

6.7 TIMER / COUNTER

Definiție

Un Timer / Counter este un bloc hardware periferic, integrat în microcontroler, a cărui funcție principală este să numere. El funcționează independent de unitatea centrală de procesare (CPU), permițând microcontrolerului să execute alte sarcini în paralel.

ATmega1280 dispune de **6 timere** dintre care 2 sunt pe **8 biți** (**Timer / Counter0**, **Timer / Counter2**) și 4 pe **16 biți** (**Timer / Counter1**, **Timer / Counter3**, **Timer / Counter4**, **TimerCounter5**).

Timer / Counterele sunt esențiale pentru realizarea sarcinilor precum:

- **managementul evenimentelor** - executare de cod la intervale precise de timp;
- **generare de unde** - crearea de semnale **PWM** (**Pulse Width Modulation**) sau cu frecvență variabilă;
- **măsurarea semnalelor** - calcularea **frecvenței**, a **factorului de umplere** (**duty cycle**) sau a **duratei** impulsurilor externe.

6.7.1 TIMER / COUNTER PE 8 BIȚI

6.7.1.1 TIMER/COUNTER0

Primul modul de temporizare **Timer/Counter0 (TCNT0)** al microcontrolerului **ATmega1280** are două registre pe **8 biți** de comparare a ieșirilor (**Output Compare Registers**) și poate genera frecvențe și diverse forme de undă, inclusiv modularea în lățime a pulsului (**Pulse Width Modulation** sau **PWM**) cu perioadă variabilă.

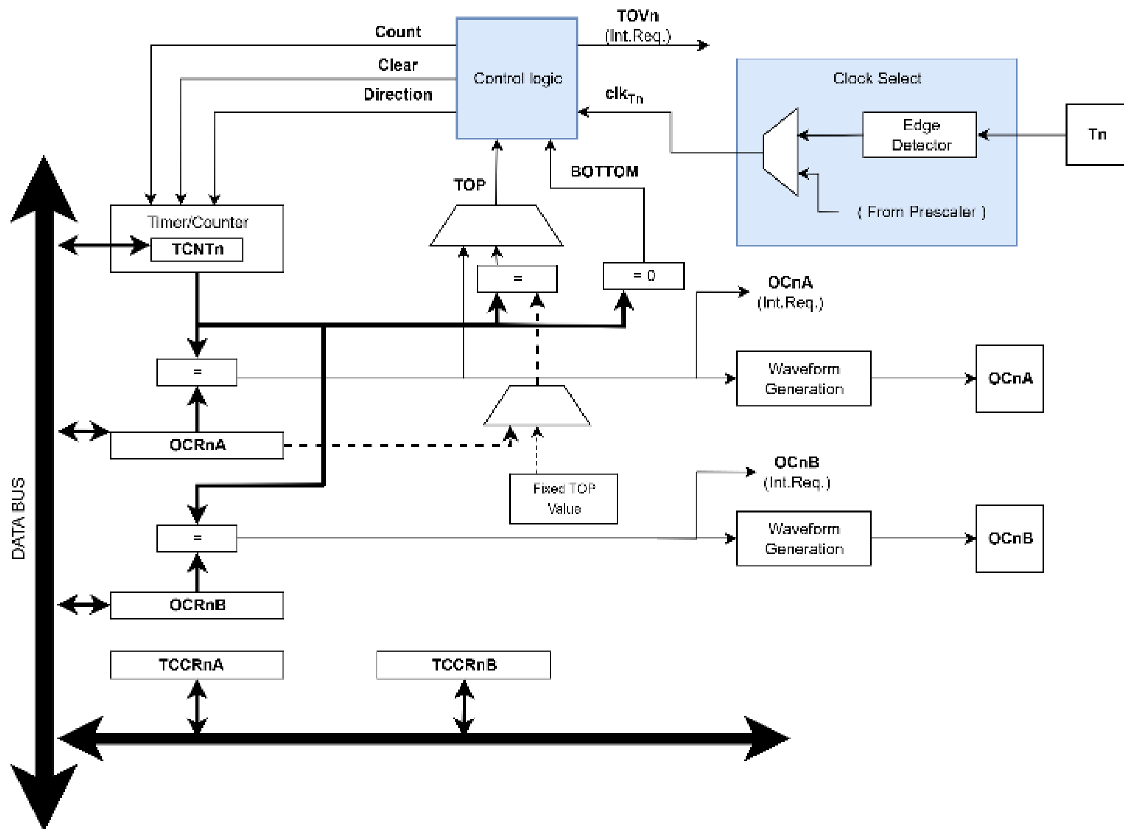


Figura 6.5 – Diagrama bloc pentru TCNT0

6.7.1.2 TIMER/COUNTER2

Al doilea modul pe **8 biți**, **Timer/Counter2 (TCNT2)**, oferă în plus capacitatea de a opera **asincron**, însă **clock-ul** este selectat diferit, deoarece trebuie să fie utilizată o sursă de clock **asincronă**.

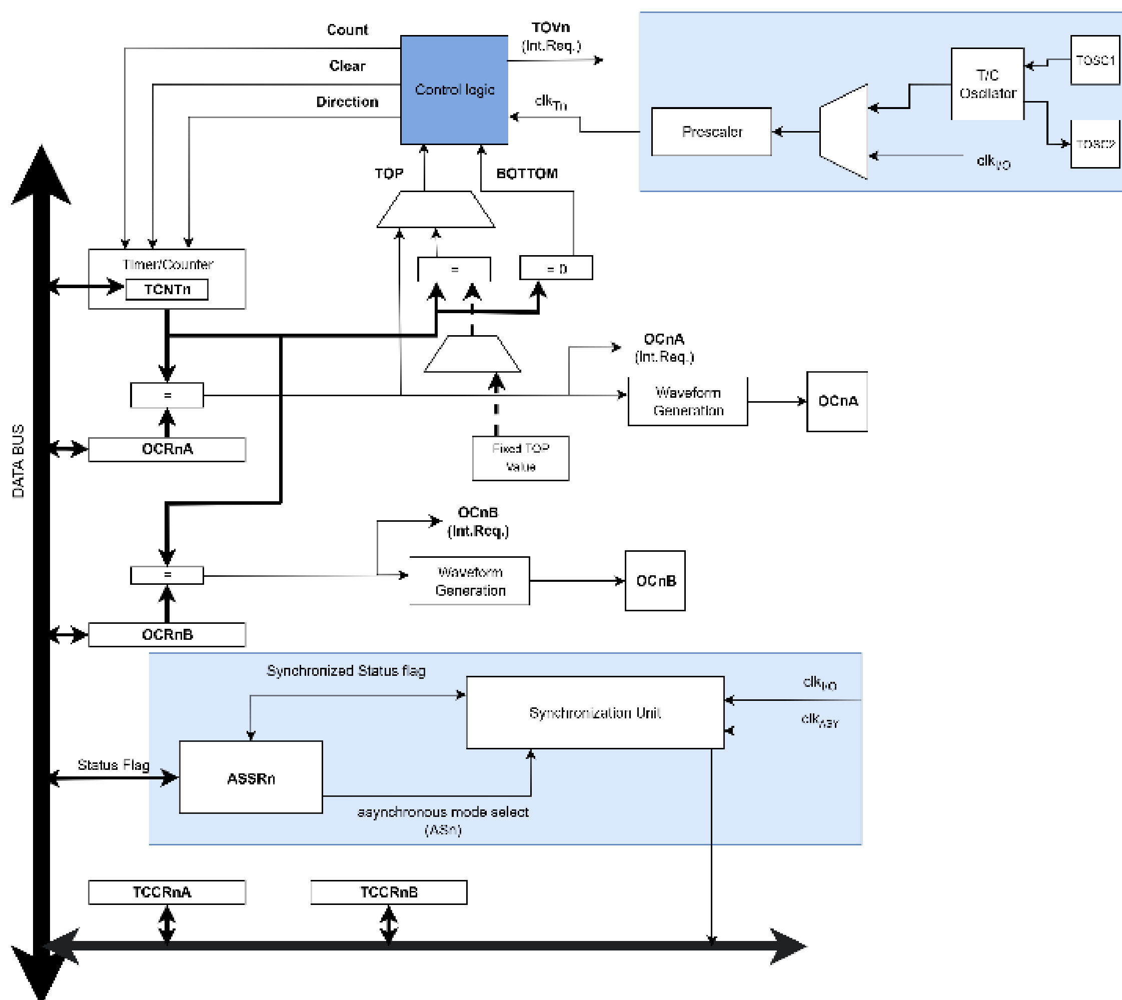


Figura 6.6 – Diagrama bloc pentru TCNT2

6.7.2 TIMER / COUNTER PE 16 BIȚI

Modulele **Timer / Counter** pe **16 biți** sunt **TCNT1**, **TCNT3**, **TCNT4** și **TCNT5**. Acestea au în plus față de **Timer / Counterele** pe **8 biți** încă un registru de comparare a ieșirii (**OCRnC**) și unitatea de captare a intrărilor (**Input Capture**) cu registrele aferente.

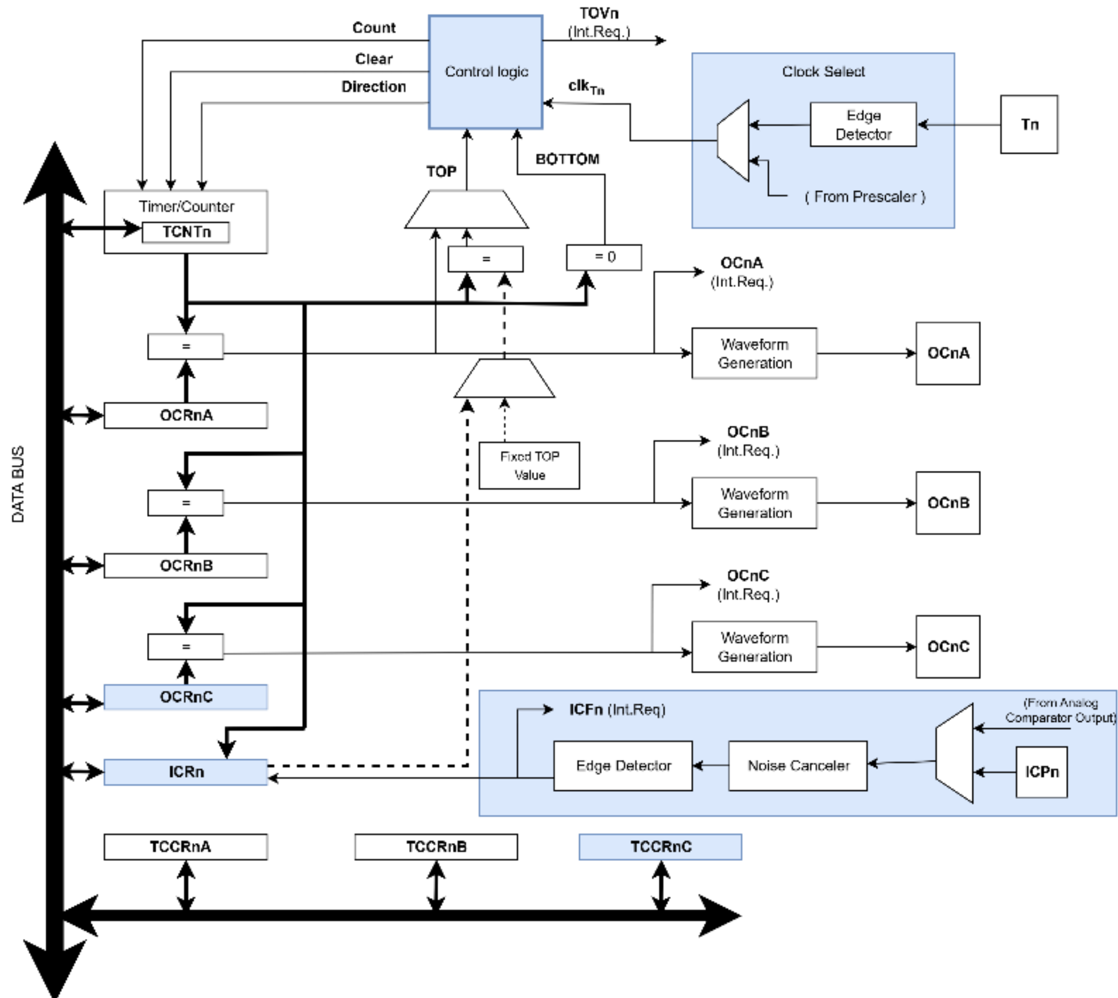


Figura 6.7 – Diagrama bloc TCNTn pe 16 biți

6.8 SURSA DE CLOCK ȘI CONTROLUL FRECVENȚEI

Sursa semnalului de ceas (**clock**) pentru modulul **Timer / Counter** poate fi selectată fie intern, derivată din ceasul de sistem (**f_clk_I/O**), fie extern, prin intermediul pinului **Tn**. La utilizarea sursei interne, aceasta poate fi aplicată direct sau divizată printr-un factor de prescalare programabil.

Controlul granular al acestor opțiuni se realizează prin configurarea biților de selecție a ceasului **(CSn2:0)** din registrul de control **TCCRnB**. Acești biți determină:

- sursa de ceas (**internă** sau **externă**).
- factorul de prescalare (pentru sursa internă).
- frontul activ – **crescător** sau **descrescător** (pentru sursa externă).

Semnalul rezultat din blocul de selecție a ceasului, la care operează efectiv temporizatorul, este denumit **clk_Tn**. În cazul în care nu este selectată nicio sursă de ceas (biții **CSn2:0** sunt setați la **0**), temporizatorul este **inactiv**, oprind orice operațiune de numărare.

6.8.1 SURSA INTERNĂ DE CLOCK

Sursa de ceas internă este derivată din ceasul de sistem al microcontrolerului (**f_clk_I/O**). Această opțiune asigură o funcționare sincronă cu restul perifericelor și este modul de operare principal pentru funcții de temporizare și generare de semnale **PWM**.

Temporizatorul (**timer**) poate fi configurat să utilizeze **direct** ceasul intern al sistemului, fără nicio divizare. În acest caz, frecvența **maximă** de operare a temporizatorului este **egală** cu frecvența sistemului (**f_clk_I/O**). Această sursă de ceas oferă **cea mai rapidă** operare și este potrivită pentru aplicațiile care necesită precizie ridicată și viteză maximă.

6.8.2 PRESCALAREA SURSEI DE CLOCK

Pentru a obține perioade de numărare mai lungi sau pentru a **reduce** frecvența de operare a temporizatorului, se poate utiliza un **prescaler**. Acesta este un **divizor de frecvență programabil** care se aplică sursei de ceas **interne**. Modulele **Timer/Counter0**, **1**, **3**, **4** și **5** partajează același modul de prescalare, dar pot avea setări diferite.

Diviziunile disponibile pentru frecvența de ceas sunt:

- **clkIO / 1 (fără prescalare);**
- **clkIO / 8;**
- **clkIO / 64;**
- **clkIO / 256;**
- **clkIO / 1024;**

Notă: Resetarea prescaler-ului (prin registrul GTCCR) va afecta toate temporizatoarele care îl utilizează, fiind necesară o resincronizare a acestora dacă operarea simultană este critică.

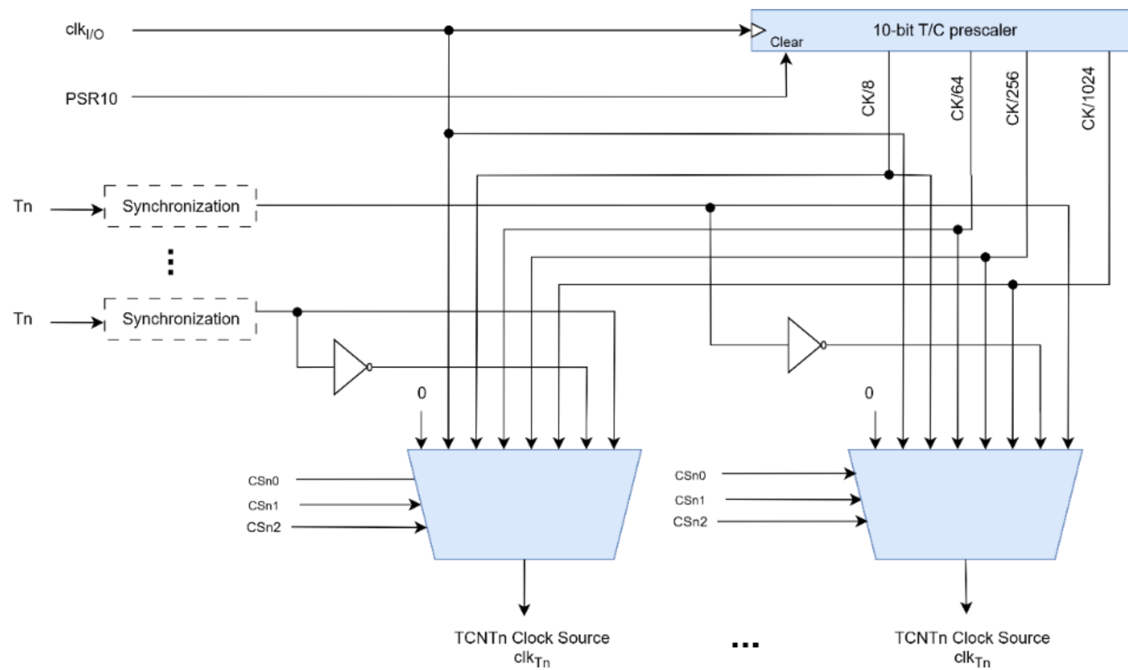


Figura 6.8 – Diagrama bloc pentru Prescaler

6.8.3 SURSA DE CLOCK EXTERNĂ

Temporizatorul poate fi configurat să utilizeze o sursă **externă** de ceas, aplicată pe pinul corespunzător **Tn** (ex: **T0**, **T1**). Această funcționalitate este esențială pentru contorizarea evenimentelor **externe**, **independent** de frecvența de operare a microcontrolerului.

Configurarea permite selectarea frontului pe care temporizatorul va acționa:

- front crescător (**rising edge**) al semnalului extern;
- front descrescător (**falling edge**) al semnalului extern.

Notă: Sursa de ceas externă **nu poate fi divizată** de către prescalerul intern.

6.8.4 OPRIREA TEMPORIZATORULUI

Temporizatorul poate fi oprit complet prin selectarea opțiunii **"No clock source"** (fără sursă de ceas), corespunzătoare valorii **0** (**000**) pentru biții **CSn2:0**. În această stare, contorul își păstrează **ultima valoare**, iar registrele asociate (**TCNTn**, **OCRnx**, etc.) rămân accesibile pentru operațiuni de **citire/scriere** din partea **CPU**, dar nicio activitate de numărare nu are loc.

6.8.5 MOD ASINCRON

După cum este menționat, modulul **Timer/Counter2 (TCNT2)** dispune de o caracteristică în plus față de celelalte temporizatoare: capacitatea de a funcționa într-un mod **asincron** față de ceasul principal al sistemului (**f_clk_I/O**). Această funcționalitate este concepută pentru aplicații de tip **Real-Time Counter (RTC)**, care necesită o bază de timp **constantă**, **independentă** de starea sau frecvența de operare a microcontrolerului.

Când modul **asincron** este activat, sursa de ceas pentru **Timer/Counter2** nu mai este derivată din ceasul de sistem, ci dintr-un **oscilator extern de joasă frecvență**, conectat la pinii **TOSC1** și **TOSC2**. Acest mod este activat prin setarea bitului **AS2** din registrul de stare asincronă (**ASSR**). Când acest bit este setat, pinii **TOSC1** și **TOSC2** sunt deconectați de la funcționalitatea lor de port **I/O standard** și devin **pinii oscilatorului dedicat**.

Notă: Un circuit electronic care produce la ieșirea sa un semnal electric periodic este cunoscut drept generator de semnale (**oscilator**). Dacă frecvența semnalului generat se încadrează în domeniul **30-300 kHz**, atunci este vorba de oscilații de **joasă frecvență**. Un oscilator se poate obține dintr-un amplificator cu reacție **pozitivă**.

De asemenea, utilizatorii trebuie să fie foarte atenți la:
1. **Comutarea între moduri**: comutarea dinamică între modul **sincron** și cel **asincron** (prin modificarea bitului **AS2**) este **periculoasă** și poate **corupe** valorile din registrele temporizatorului.
2. **Frecvența sistemului**: ceasul principal al sistemului (**f_clk_I/O**) trebuie să fie de **cel puțin 4 ori mai mare** decât **frecvența oscilatorului** extern pentru a asigura o eșantionare **corectă** a stărilor interne.
3. **Timpul de stabilizare al oscilatorului**: după un **Power-up Reset** sau o ieșire din modul **Power-down**, oscilatorul extern poate necesita până la o secundă pentru a se stabiliza. Utilizatorul este sfătuit să aștepte acest interval înainte de a utiliza temporizatorul pentru operațiuni critice.

6.9 UNITATEA DE NUMĂRARE

Unitatea de numărare se ocupă de **incrementarea** / **decrementarea** propriu-zisă a valorii din registrul **TCNTn** și este controlată prin semnalele **TOP** (valoare **maximă** atinsă) și **BOTTOM** (valoare **minimă** atinsă). Registrul **TCNTn** este conectat la magistrala de date pe **8 biți**; în cazul **Timer / Counterelor** pe **16 biți**, acesta este format din 2 registre pe **8 biți** (**TCNTnH** și **TCNTnL**) și este **scris** / **citit** prin 2 operații și folosirea unui registru temporar.

Semnalele interne au următoarele funcționalități:
- **count**: incrementează sau decrementează valoarea din registrul **TCNTn** cu **1**, în funcție de configurație;
- **direction**: selectează direcția de numărare (**incrementare** sau **decrementare**);
- **clear**: șterge conținutul registrului **TCNTn**, setând toți biții acestuia la **0**;
- **clkTn**: semnalul de clock pentru **Timer / Counter**, la fiecare front **activ** efectuându-se o operație de numărare;
- **top**: semnalează faptul că registrul **TCNTn** a atins valoarea sa **maximă** (dacă este setată la valoarea **MAX**, **0xFF** pe **8 biți** sau **0xFFFF** pe **16 biți**);
- **bottom**: semnalează faptul că registrul **TCNTn** a atins valoarea sa **minimă** (**0**).

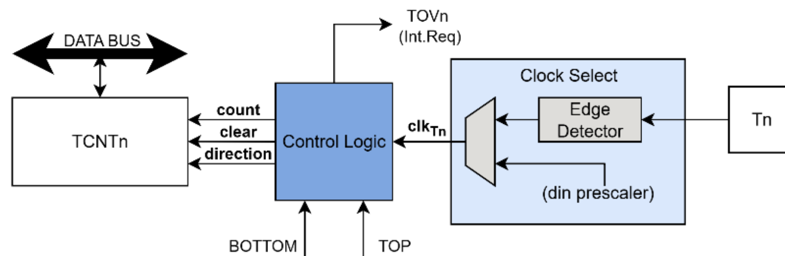


Figura 6.9 – Diagrama bloc pentru unitatea de măsurare TCNT0

Diagrama bloc a unității de numărare pentru **Timer/Counter2** include două oscilatoare **TOSC1** și **TOSC2** pentru selecția **clock-ului**, întrucât acestea sunt de frecvență mult mai joasă decât ceasul principal și pot funcționa **asincron** față de **CPU**.

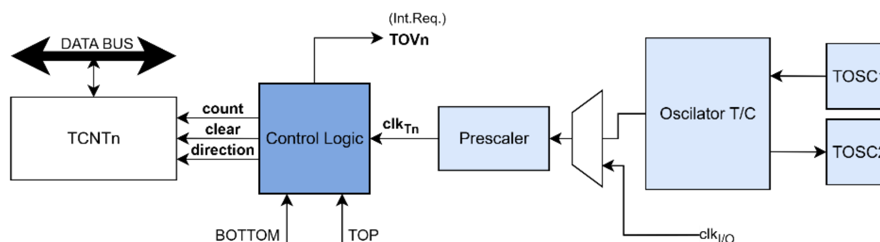


Figura 6.10 – Diagrama bloc pentru unitatea de numărare TCNT2

Timer / Counterele pe **16 biți** au 2 registre de **8 biți** (**TCNTnH și TCNTnL**) pentru a stoca valoarea numărată.

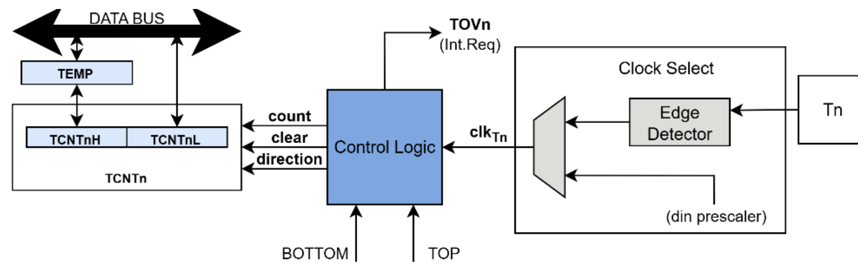


Figura 6.11 – Diagrama bloc pentru unitatea de numărare TCNTn pe 16 biți

6.10 UNITATEA DE COMPARARE

Această unitate compară **continuu TCNTn** cu **OCRnx**. Dacă **TCNTn** este **egal** cu registrul **OCRnx** se va seta flagul **Output Compare (OCFnx)** la următorul ciclu de ceas. Acest flag va genera o întrerupere de **output compare** dacă această întrerupere este activată (**OCIEnx = 1**). Flagul **Output Compare** (***OCFnx***) este șters automat când întreruperea este executată sau poate fi resetat prin intermediul software-ului prin scrierea la locația sa un **1 logic**. Generatorul semnalului de ieșire (**Waveform Generator**) folosește semnalul de ieșire al comparatorului pentru a **genera** o formă de undă corespunzătoare cu modul de operare setat prin biții **WGMn3:0** și biții de **Compare Output Mode (COMnx1:0)**.

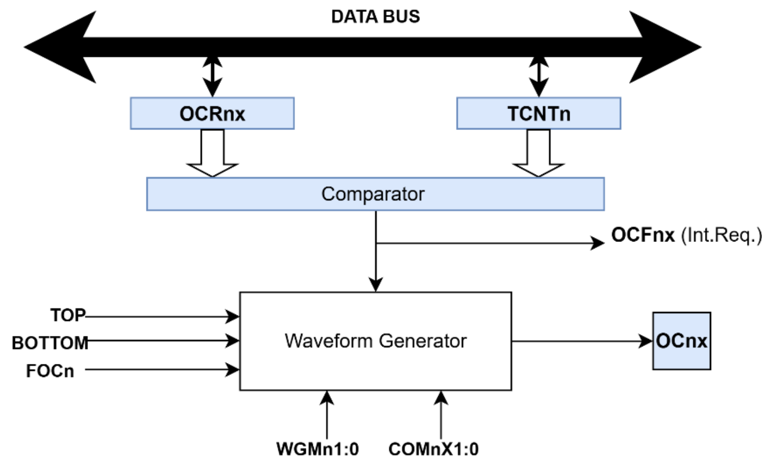


Figura 6.12 – Diagrama bloc pentru unitatea de comparare TCNTn pe 8 biți

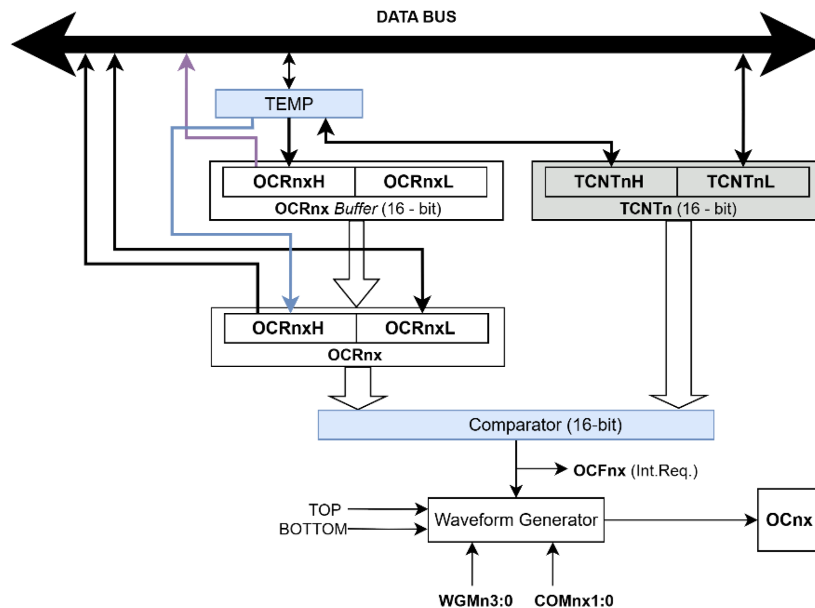


Figura 6.13 – Diagrama bloc pentru unitatea de comparare TCNTn pe 16 biți

6.11 UNITATEA INPUT CAPTURE PENTRU TCNTN 16-BIT

Temporizatoarele încorporează o unitate de **captare a intrărilor** care poate captura evenimente externe cărora le poate oferi un ***timestamp*** la momentul apariției. Semnalul extern care indică un eveniment (sau mai multe evenimente) poate fi aplicat prin intermediul pinului **ICPn** sau, alternativ, numai pentru **TCNT1**, prin **Analog Comparator Unit**. Timestamp-ul poate fi utilizat pentru a **calcula frecvența**, **factorul de umplere** (**duty cycle**) și alte caracteristici ale semnalului generat sau pentru a crea un jurnal al evenimentelor.

Când are loc un eveniment (se schimbă nivelul logic) pe pinul **Input Capture (ICPn)**, alternativ pe ieșirea **AC (Analog Comparator)**, iar această modificare confirmă setarea **detectorului de front** (**edge detector**), se va declanșa o captură. În momentul declanșării unei capturi, valoarea **TCNT** este scrisă în registrul **ICRn**, iar flagul corespunzător capturii **(ICFn)** este setat la același **clock**. Dacă este activat (**TICIEn = 1**), atunci flagul **ICFn** generează o întrerupere, iar valoarea sa este ștearsă automat când este executată întreruperea.

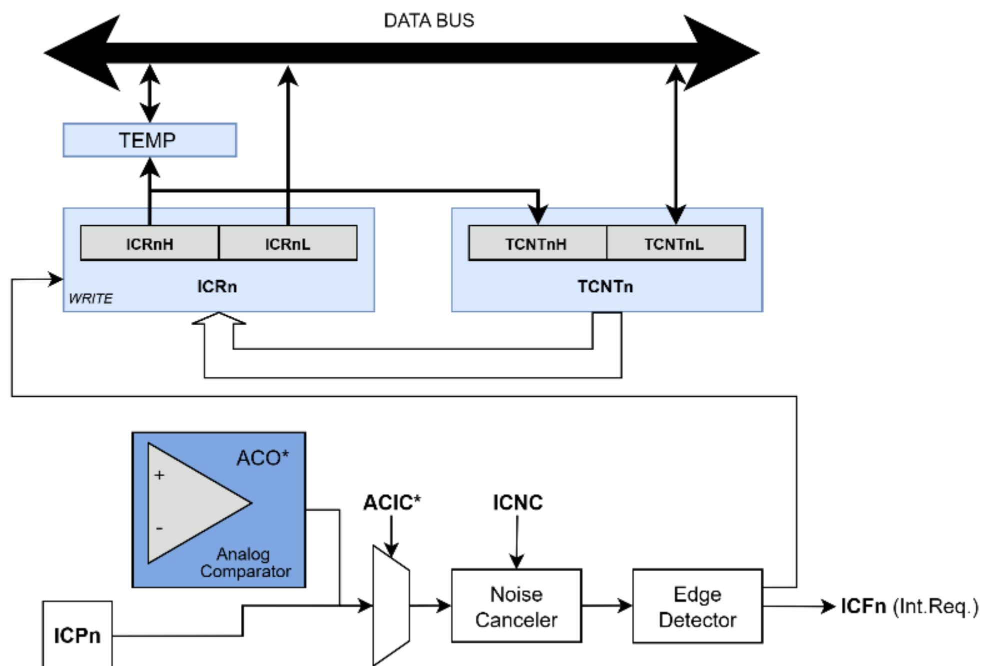


Figura 6.14 – Diagrama bloc pentru unitatea Input Capture TCNTn pe 16 biți

6.12 REGIȘTRI

6.12.1 TEMPORIZATOARE PE 8 BIȚI

6.12.1.1 TCCRNA – TIMER / COUNTER CONTROL REGISTER A

Bit	7	6	5	4	3	2	1	0	
0x24 (0x44)	COM0A1	COM0A0	COM0B1	COM0B1	-	-	WGM01	WGM00	TCCR0A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.15 - Registrul TCCR0A

- **Biții 7:6 – COMnA1:0: Compare Match Output A Mode**
  Sunt biții de control pentru pinul de **Output Compare** (**OCnA**). Dacă cel puțin un bit dintre aceștia este setat, ieșirea **OCnA** va **suprascrie** funcționalitatea portului pinului **I/O** la care este conectat.

COM0A1	COM0A0	Descriere
0	0	Funcționarea normală a portului, **OC0A** deconectat
0	1	Activează **OC0A** la Compare Match
1	0	Șterge **OC0A** la Compare Match
1	1	Setează **OC0A** la Compare Match

Tabelul 6.1 - Compare Output Mode, non-PWM Mode

COM0A1	COM0A0	Descriere
0	0	Funcționare normală a portului, **OC0A** deconectat
0	1	Când **WGM02 = 0**: funcționare normală a portului, **OC0A** deconectat; Când **WGM02 = 1**: se activează **OC0A** la Compare Match
1	0	Șterge **OC0A** la Compare Match, setează **OC0A** la valoarea **BOTTOM** (modul neinversat)
1	1	Setează **OC0A** la Compare Match, șterge **OC0A** când ajunge la valoarea **BOTTOM** (modul inversat)

Tabelul 6.2 - Compare Output Mode, Fast PWM Mode

COM0A1	C0M0A0	Descriere
0	0	Funcționare normală a portului, **OC0A** deconectat
0	1	Când **WGM02 = 0**: funcționare normală a portului, **OC0A** deconectat; Când **WGM02 = 1**: se activează **OC0A** la Compare Match
1	0	În momentul incrementării se va șterge **OC0A** la Compare Match. În momentul decrementării, se va seta **OC0A** la Compare Match.
1	1	În momentul incrementării se va seta **OC0A** la Compare Match. În momentul decrementării, se va șterge **OC0A** la Compare Match.

Tabelul 6.3 - Compare Output Mode, Phase Correct PWM Mode

- **Biții 5:4 – C0MnB1:0 Compare Match Output Mode**
  Sunt biții de control pentru pinul de **Output Compare** (**OCnB**). Dacă cel puțin un bit dintre aceștia este setat, ieșirea **OCnB** va suprascrie funcționalitatea portului pinului I/O la care este conectat.

COM0B1	COM0B0	Descriere
0	0	Funcționarea normală a portului, **OC0B** deconectat
0	1	Activează **OC0B** la Compare Match
1	0	Șterge **OC0B** la Compare Match
1	1	Setează **OC0B** la Compare Match

Tabelul 6.4 - Compare Output Mode, non-PWM Mode

COM0B1	COM0B0	Descriere
0	0	Funcționare normală a portului, **OC0B** deconectat
0	1	Rezervat
1	0	Șterge **OC0B** la Compare Match, setează **OC0B** la valoarea **BOTTOM** (modul neinversat)
1	1	Setează **OC0B** la Compare Match, Șterge **OC0B** când ajunge la valoarea **BOTTOM** (modul inversat)

Tabelul 6.5 - Compare Output Mode, Fast PWM Mode

COM0B1	C0M0B0	Descriere
0	0	Funcționare normală a portului, **OC0B** deconectat
0	1	Rezervat
1	0	În momentul incrementării se va șterge **OC0B** la Compare Match. În momentul decrementării, se va seta **OC0B** la Compare Match.
1	1	În momentul incrementării se va seta **OC0B** la Compare Match. În momentul decrementării, se va șterge **OC0B** la Compare Match.

Tabelul 6.6 - Compare Output Mode, Phase Correct PWM Mode

- **Biții 3:2 – Res (Reserved Bits)**
  Acești biți sunt rezervați și mereu vor avea valoarea **0** la citire.

- **Biții 1:0 – WGMn1:0 Waveform Generation Mode**
  Împreună cu bitul **WGMn2** din registrul **TCCRnB**, acești biți controlează secvența de numărare a contorului, sursa pentru valoarea maximă (**TOP**) a contorului și **tipul de generare** a formei de undă care va fi utilizat.

Mode	WGM2	WGM1	WGM0	Timer / Counter Mode of Operation	TOP	Update of OCRx at	TOV Flag Set on
0	0	0	0	Normal	0xFF	Immediate	MAX
1	0	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	0	1	0	CTC	OCRA	Immediate	MAX
3	1	1	1	Fast PWM	0xFF	TOP	MAX
4	1	0	0	Reserved	-	-	-
5	1	0	1	PWM, Phase Correct	OCRA	TOP	BOTTOM
6	1	1	0	Reserved	-	-	-
7	1	1	1	Fast PWM	OCRA	BOTTOM	TOP

Tabelul 6.7 - Moduri de operare pentru TCNT0

Notă: Modul Reserved nu este un mod de funcționare propriu-zis. Reserved (Rezervat) indică o combinație specifică a biților de configurare pe care producătorul a decis să nu o documenteze pentru uz public.

6.12.1.2 TCCRNB – TIMER / COUNTER CONTROL REGISTER B

Bit	7	6	5	4	3	2	1	0	
0x25 (0x45)	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00	TCCR0B
Read/Write	W	W	R	R	R/W	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.16 - Registrul TCCRNB

- **Bitul 7 – FOCnA: Force Output Compare A**
  Acest bit este activ doar atunci când biții **WGM** specifică modul **non-PWM**, este implementat ca un impuls și va avea mereu valoarea **0**.

- **Bitul 6 – FOCnB: Force Output Compare B**
  Acest bit este activ doar atunci când biții **WGM** specifică modul **non-PWM**, este implementat ca un impuls și va avea mereu valoarea **0**.

- **Biții 5:4 – Reserved Bits**

- **Bitul 3 – WGMn2: Waveform Generation Mode**

- **Biții 2:0 – CSn2:0: Clock Select**
  Acești biți selectează **sursa de ceas** utilizată de temporizator.

CS02	CS01	CS00	Descriere
0	0	0	No clock source – temporizatorul este oprit
0	0	1	$clk_{I/O}$ (fără prescaler)
0	1	0	$clk_{I/O}$ / 8 (from prescaler)
0	1	1	$clk_{I/0}$ / 64 (from prescaler)
1	0	0	$clk_{I/O}$ / 256 (from prescaler)
1	0	1	$clk_{I/O}$ / 1024 (from prescaler)
1	1	0	Sursa externă de clock prin intermediul pinului **T0** pentru front descrescător
1	1	1	Sursa externă de clock prin intermediul pinului **T0** pentru front crescător

Tabelul 6.8 – Descrierea biților de Clock Select

6.12.1.3 TCNTN – TIMER / COUNTER REGISTER

Registrul principal care numără, poate fi tactat intern prin intermediul unui **prescaler** sau printr-o sursă **externă de clock** prin intermediul pinului **Tn**.

Bit	7	6	5	4	3	2	1	0	
0x26 (0x46)	TCNT0[7:0]								**TCNT0**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.17 - Registrul TCNT0

6.12.1.4 OCRNA – OUTPUT COMPARE REGISTER A

Conține o valoare pe **8 biți** care este comparată continuu cu valoarea contorului (**TCNTn**). O potrivire (**match**) poate fi folosită pentru a genera întreruperi **Output Compare** sau pentru a genera la ieșire prin pinul **OCnA** o formă de undă.

Bit	7	6	5	4	3	2	1	0	
0x27 (0x47)	OCR0A[7:0]								**OCR0A**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.18 - Registrul OCR0A

6.12.1.5 OCRNB – OUTPUT COMPARE REGISTER B

Conține o valoare pe **8 biți** care este comparată continuu cu valoarea contorului (**TCNTn**). O potrivire (**match**) poate fi folosită pentru a genera întreruperi **Output Compare** sau pentru a genera la ieșire prin pinul OCnB o formă de undă.

Bit	7	6	5	4	3	2	1	0	
0x28 (0x48)	OCR0B[7:0]								**OCR0B**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.19 - Registrul OCR0B

6.12.1.6 TIMSKN – TIMER / COUNTER INTERRUPT MASK REGISTER

Bit	7	6	5	4	3	2	1	0	
(0x6E)	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0	**TIMSK0**
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.20 - Registrul TIMSK0

- **Biții 7:3 – Res: Reserved Bits.**

- **Bit 2 – OCIEnB: Timer / Counter Output Compare Match B Interrupt Enable**
  Atunci când bitul **OCIEnB** este setat pe valoarea **1 logic** și bitul **I** din registrul de stare este **activat**, se activează întreruperea **Timer / Counter Compare Match B**. Întreruperea corespunzătoare se va executa atunci când bitul **OCFnB** este **activat**, adică atunci când **OCRnB = TCNTn**.

- **Bitul 1 – OCIEnA: Timer / Counter Output Compare Match A Interrupt Enable - Idem**.

- **Bitul 0 – TOIEn: Timer / Counter Overflow Interrupt Enable**
  Atunci când **TOIEn** este setat pe valoarea **1 logic** și bitul **I** din registrul de stare este activat, se activează întreruperea **Timer / Counter n Overflow Interrupt**. Întreruperea corespunzătoarea se va executa atunci când contorul depășește valoarea **MAX** (**0xFF**) sau **BOTTOM** (**0x00**), iar bitul **TOVn** e activat.

6.12.1.7 TIFRN – TIMER / COUNTER N INTERRUPT FLAG REGISTER

Bit	7	6	5	4	3	2	1	0	
0x15(0x35)	-	-	-	-	-	OCF0B	OCF0A	TOV0	**TIFR0**
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.21 - Registrul TIFR0

- **Biții 7:3 – Res: Reserved Bits.**

- **Bitul 2 – OCFnB: Timer / Counter n Output Compare B Match Flag**
  Acest bit este **activat** atunci când **TCNTn = OCRnB**. **OCFnB** este șters de hardware atunci când se execută vectorul corespunzător de gestionare a întreruperilor.

- **Bitul 1 – OCFnA: Timer / Counter n Output Compare A Match Flag**
  Acest bit este **activat** atunci când **TCNTn = OCRnA**. **OCFnA** este șters de hardware atunci când se execută vectorul corespunzător de gestionare a întreruperilor.

- **Bitul 0 – TOVn: Timer / Counter n Overflow Flag**
  Acest bit **se activează** atunci când apare **overflow** în **TCNTn**. **TOVn** este șters de hardware atunci când se execută vectorul corespunzător de gestionare a întreruperilor.

Notă: Figurile și tabelele utilizate mai sus sunt pentru **Timer / Counter0**, însă se aplică și pentru **Timer / Counter2**, înlocuind valoarea lui **n** = **2**. Registrele **TCNTn**, **TCCRnA** (**TCCRnB** este utilizat pentru unitatea **Input Capture**), **TIMSKn**, **TIFRn**, **OCRnx** au aceeași funcționalitate și pentru temporizatoarele pe **16 biți**.

În comparație cu **TCNT0**, **TCNT2** mai are în plus regiștrii **ASSR** și **GTCCR** pentru operarea în modul **asincron.** Atenție, registrul **ASSR** este utilizat doar de **Timer / Counter2.**

6.12.1.8 ASSR – ASYNCHRONOUS STATUS REGISTER

Bit	7	6	5	4	3	2	1	0	
(0xB6)	-	EXCLK	AS2	TCN2UB	OCR2AUB	OCR2BUB	TCR2AUB	TCR2BUB	**ASSR**
Read/Write	R	R/W	R/W	R	R	R	R	R	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.22 - Registrul ASSR

- **Bitul 7 - Res: Reserved Bits.**

- **Bitul 6 – EXCLK: Enable External Clock Input**
  Atunci când bitul **EXCLK** are valoare **1 logic** și este selectat ceasul **asincron**, buffer-ul de intrare pentru ceasul extern este **activat**, acesta putând fi introdus pe pinul **Timer Oscilator 1** (**TOSC1**) în locul unui cristal de **32 kHz**. Operația de scriere în **EXCLK** ar trebui făcută înainte de a selecta modul de operare **asincron**.

- **Bitul 5 – AS2: Asynchronous Timer / Counter2**
  Când acest bit este **0**, **TCNT2** utilizează sursa de ceas a sistemului. Atunci când acest bit este **1**, **TCNT2** folosește o sursă de clock **externă** conectată la pinul **TOSC1**.

- **Bitul 4 – TCN2UB: Timer / Counter2 Update Busy**
  Atunci când **TCNT2** funcționează în modul **asincron** și **TCNT2** este scris se activează acest bit. Dacă **TCNT2** a fost actualizat dintr-un registru de stocare temporar, acest bit va fi **șters de hardware**. Valoarea **0 logic** indică că **TCNT2** este „pregătit" să fie **actualizat** cu o valoare nouă.

- **Bitul 3 – OCR2AUB: Output Compare Register2 Update Busy – Idem.**

- **Bitul 2 – OCR2BUB: Output Compare Register2 Update Busy – Idem.**

- **Bitul 1 - TCR2AUB: Timer / Counter Control Register2 Update Busy – Idem.**

- **Bitul 0 - TCR2BUB: Timer / Counter Control Register2 Update Busy – Idem.**

6.12.1.9 GTCCR – GENERAL TIMER / COUNTER CONTROL REGISTER

Bit	7	6	5	4	3	2	1	0	
0x23 (0x43)	TSM	-	-	-	-	-	PSRASY	PSRSYNC	GTCCR
Read/Write	R/W	R	R	R	R	R	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.23 - Registrul GTCCR

- **Bitul 7 – TSM: Timer / Counter Synchronization Mode**
  Atunci când valoarea acestui bit este **1 logic** se activează modul **sincron** al temporizatorului. Când acest bit este setat pe valoarea **0 logic**, contoarele încep să numere **simultan**.

- **Bitul 1 – PSRASY: Prescaler Reset Timer / Counter2**
  Atunci când acest bit are valoarea **1 logic**, factorul de scalare pentru **TCNT2** va fi **resetat**.

- **Bitul 0 – PSRSYNC: Prescaler Reset for Synchronous Timer / Counter**
  Atunci când acest bit este setat pe valoarea **1 logic**, valoarea factorului de scalare pentru **TCNT0**, **TCNT1**, **TCNT3**, **TCNT4** și **TCNT5** va fi **resetată**.

6.12.2 TEMPORIZATOARE PE 16 BIȚI

6.12.2.1 TCCRNA – TIMER / COUNTER N CONTROL REGISTER A

Bit	7	6	5	4	3	2	1	0	
	COMnA1	COMnA0	COMnB1	COMnB1	COMnC1	COMnC0	WGMn1	WGMn0	TCCRnA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.24 - Registrul TCCRnA

- **Biții 7:6 – COMnA1:0: Compare Output Mode for Channel A**

- **Biții 5:4 – COMnB1:0: Compare Output Mode for Channel B**

- **Biții 3:2 – COMnC1:0: Compare Output Mode for Channel C**
  Acești biți sunt biții de control pentru pinii **Output Compare** (**OCnA**, **OCnB**, **OCnC**). Dacă cel puțin un bit dintre aceștia este setat, ieșirea **OCnA** va suprascrie funcționalitatea portului pinului **I/O** la care este conectat.

- **Biții 1:0 – WGMn1:0: Waveform Generation Mode**
  Împreună cu bitul **WGMn3:2** din registrul **TCCRnB**, acești biți controlează secvența de numărare a contorului, sursa pentru valoarea maximă (**TOP**) a contorului și tipul de generare a formei de undă care va fi utilizat.

COMnA1 COMnB1 COMnC1	COM0A0 COM0B0 COM0C0	Descriere
0	0	Funcționarea normală a portului, **OCnA** / **OCnB** / **OCnC** deconectat
0	1	Activează **OCnA** / **OCnB** / **OCnC** la Compare Match
1	0	Șterge **OCnA** / **OCnB** / **OCnC** la Compare Match (setează ieșirea pe **0** logic)
1	1	Setează **OCnA** / **OCnB** / **OCnC** la Compare Match (setează ieșirea pe **1** logic)

Tabelul 6.9 – Compare Output Mode, non-PWM

COMnA1 COMnB1 COMnC1	COMnA0 COMnB0 COMnC0	Descriere
0	0	Funcționare normală a portului, **OCnA** / **OCnB** / **OCnC** deconectat
0	1	Când **WGM13:0 = 14** sau **15**: se activează **OC1A** la Compare Match și **OC1B** și **OC1C** sunt deconectate. Pentru celelalte setări ale pinilor **WGM1** portul va funcționa normal, iar **OCnA** / **OCnB** / **OCnC** sunt deconectate
1	0	Șterge **OCnA** / **OCnB** / **OCnC** la Compare Match, setează **OCnA** / **OCnB**/ **OCnC** la valoarea **BOTTOM** (modul neinversat)
1	1	Setează **OCnA** / **OCnB** / **OCnC** la Compare Match, Șterge **OCnA** / **OCnB** / **OCnC** când ajunge la valoarea **BOTTOM** (modul inversat)

Tabelul 6.10 – Compare Output Mode, Fast PWM

COM0A1	C0M0A0	Descriere
0	0	Funcționare normală a portului, **OCnA** / **OCnB** / **OCnC** deconectat
0	1	Când **WGM13:0 = 9** sau **11**: se activează **OC1A** la Compare Match și **OC1B** și **OC1C** sunt deconectate. Pentru celelalte setări ale pinilor **WGM1** portul va funcționa normal, iar **OCnA** / **OcnB** / **OCnC** sunt deconectate
1	0	În momentul incrementării se va șterge **OCnA** / **OCnB** / **OCnC** la Compare Match. În momentul decrementării, se va seta **OCnA** / **OCnB** / **OCnC** la Compare Match.
1	1	În momentul incrementării se va seta **OCnA** / **OCnB** / **OCnC** la Compare Match. În momentul decrementării, se va șterge **OCnA** / **OCnB** / **OCnC** la Compare Match.

Tabelul 6.11 – Compare Output Mode, Phase Correct și Phase and Frequency Correct PWM

6.12.2.2 TCCRNB – TIMER / COUNTER N CONTROL REGISTER B

Bit	7	6	5	4	3	2	1	0	
	ICNCn	ICESn	-	WGMn3	WGMn2	CSn2	CSn1	CSn0	**TCCRnB**
Read/Write	W	W	R	R/W	R/W	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.25 - Registrul TCCRnB

- **Bitul 7 – ICNCn: Input Capture Noise Canceler**
  Este activ când are valoarea **1 logic**. Când **Noise Canceler** este activ, semnalul de la pinul **Input Capture** (**ICPn**) este filtrat. Funcția de filtrare necesită **patru eșantioane** succesive **egale** ale pinului **ICPn** pentru a schimba valoarea la ieșire. Prin urmare, captura semnalului va fi întârziată cu **patru ciclii** ale oscilatorului atunci când **Noise Canceler** este **activ**.

- **Bitul 6 – ICESn: Input Capture Edge Select**
  Acest bit permite selectarea frontului care va fi utilizat pentru declanșarea evenimentului de captură pe **Input Capture Pin** (**ICPn**). Când bitul **ICESn = 0**, se va utiliza frontul descrescător pentru a declanșa captura. Când bitul **ICESn = 1**, se va utiliza frontul crescător pentru a declanșa captura.

- **Bitul 5 – Res: Reserved**.

- **Biții 4:3 – WGMn3:2: Waveform Generation Mode**.

- **Biții 2:0 – CSn2:0: Clock Select**.

CS02	CS01	CS00	Descriere
0	0	0	No clock source – temporizatorul este oprit
0	0	1	$clk_{I/O}$ (fără prescaler)
0	1	0	$clk_{I/O}$ / 8 (from prescaler)
0	1	1	$clk_{I/0}$ / 64 (from prescaler)
1	0	0	$clk_{I/O}$ / 256 (from prescaler)
1	0	1	$clk_{I/O}$ / 1024 (from prescaler)
1	1	0	Sursa externă de clock prin intermediul pinului Tn pentru front descrescător
1	1	1	Sursa externă de clock prin intermediul pinului Tn pentru front crescător

Tabelul 6.12 – Descrierea bitului Clock Select

Mode	WGMn3	WGMn2 (CTC)	WGMn1 (PWMn1)	WGMn0 (PWMn0)	Time / Counter Mode of Operation	TOP	Update of OCRnx at	TOVn Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCRnA	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICRn	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCRnA	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICRn	TOP	BOTTOM

11	1	0	1	1	PWM, Phase Correct	OCRnA	TOP	BOTTOM
12	1	1	0	0	CTC	ICRn	Immediate	MAX
13	1	1	0	1	(Reserved)	-	-	-
14	1	1	1	0	Fast PWM	ICRn	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCRnA	BOTTOM	TOP

Tabelul 6.13 – Moduri de operare pentru TCNT1, 3, 4 și 5

6.12.2.3 TCCRNC – TIMER / COUNTER N CONTROL REGISTER C

Bit	7	6	5	4	3	2	1	0	
	FOCnA	FOCnB	FOCnC	-	-	-	-	-	TCCRnC
Read/Write	W	W	W	R	R	R	R	R	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.26 - Registrul TCCRnC

- **Bitul 7 – FOCnA: Force Output Compare for Channel A**
- **Bitul 6 – FOCnB: Force Output Compare for Channel B**
- **Bitul 5 – FOCnC: Force Output Compare for Channel C**
- **Biții 4:0 – Res: Reserved.**

Biții **FOCnA** / **FOCnB** / **FOCnC** sunt activi doar când **WGMn3:0** specifică un mod de operare **non-PWM**.

6.12.2.4 TCNTNH ȘI TCNTNL – TIMER / COUNTER

Bit	7	6	5	4	3	2	1	0	
	TCNTn[15:8]								TCNTnH
	TCNTn[7:0]								TCNTnL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.27 - Registrul TCNTnH și TCNTnL

6.12.2.5 OCRNXH ȘI OCRNXL – OUTPUT COMPARE REGISTER

Bit	7	6	5	4	3	2	1	0	
	OCRnx[15:8]								OCRnxH
	OCRnx[7:0]								OCRnxL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.28 - Registrul OCRnXH și OCRnXL

6.12.2.6 ICRNH ȘI ICRNL – INPUT CAPTURE REGISTER

Valoarea acestui registru este actualizată cu valoarea contorului (**TCNTn**) la fiecare eveniment al pinul **ICPn**.

Bit	7	6	5	4	3	2	1	0	
	ICRn[15:8]								ICRnH
	ICRn[7:0]								ICRnL
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.29 - Registrul ICRNH și ICRNL

6.12.2.7 TIMSKN – TIMER / COUNTER N INTERRUPT MASK REGISTER

Bit	7	6	5	4	3	2	1	0	
	-	ICIEn	-	-	OCIEnC	OCIEnB	OCIEnA	TOIEn	TIMSKn
Read/Write	R	R/W	R	R	R/W	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.30 - Registrul TIMSKn

În comparație cu **TIMSKn** pentru **TCNTn** pe **8 biți**, sunt în plus următorii biți:

- **Bitul 6 – ICIEn: Input Capture Interrupt Enable**
  Când valoarea acestui bit este **1 logic** și flagul **I** din **Status Register** este setat, întreruperea pentru Timer / Counter n **Input Capture** este activată.

- **Bitul 3 – OCIEnC: Output Compare C Match Interrupt Enable**
  Când valoarea bitului este **1 logic** și flagul **I** din **Status Register** este setat, întreruperea pentru Timer / Counter **Output Compare C Match** este activată.

6.12.2.8 TIFRN – TIMER / COUNTER N INTERRUPT FLAG REGISTER

Bit	7	6	5	4	3	2	1	0	
	-	-	ICFn	-	OCFnC	OCFnB	OCFnA	TOVn	TIFRn
Read/Write	R	R	R/W	R	R/W	R/W	R/W	R/W	
Valoare Inițială	0	0	0	0	0	0	0	0	

Figura 6.31 - Registrul TIFRn

În comparație cu **TIFRn** pentru **TCNTn** pe **8 biți**, sunt în plus următorii biți:

- **Bitul 5 – ICFn: Input Capture Interrupt Flag**
  Acest flag este setat când apare un eveniment pe pinul **ICPn**. Atunci când **ICRn** este setat de biții **WGMn3:0** să utilizeze valoarea **TOP**, flagul **ICFn** este setat când contorul atinge valoarea **TOP**. **ICFn** este șters automat când se execută vectorul de întreruperi corespunzător.

- **Bitul 3 – OCFnC: Output Compare C Match Flag**
  Acest flag este setat atunci când valoarea **TCNTn** = **OCRnC**.

6.13 MODEL PROGRAMARE

Pașii necesari pentru configurarea timer-ului:

1. **Inițializare pin de ieșire** – se setează pinul asociat canalului **PWM** ca output în registrul **DDR**.
2. **Selectarea timer-ului** – se poate alege un timer pe **8 biți** (**TCNT0**, **TCNT2**) sau pe **16 biți** (**TCNT1**, **TCNT3**, **TCNT4**, **TCNT5**).
3. **Configurarea modului de generare a formei de undă** – se setează biții **WGMn3:0** din registrele **TCCRnA** / **TCCRnB**.
4. **Setarea modului de comparare pe pin (COMnx1:0)** – se alege modul "**non-inversat**" (semnal **HIGH** la începutul perioadei și **LOW** la comparației) sau **inversat**.
5. **Alegerea sursei de clock și a prescalerului** – se configurează biții **CSn2:0** în registrul **TCCRnB** pentru prescaler (**1**, **8**, **64**, **256**, **1024**).
6. **Setarea factorului de umplere (duty cycle) –** se scrie în registrul **OCRnx**.
7. **Creare buclă principală –** se poate modifica **OCRnx** în timp real pentru a ajusta factorul de umplere.

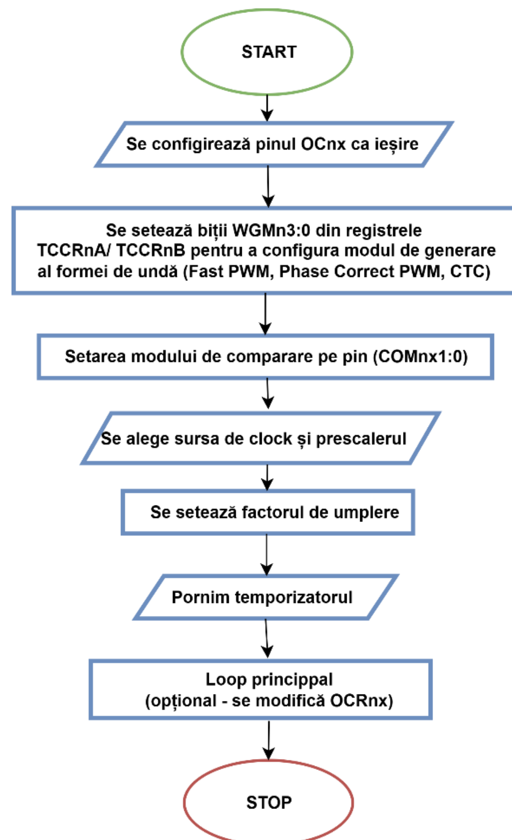


Figura 6.32 – Diagrama pentru configurarea PWM

6.14 PROBLEME

6.14.1 GENERAREA UNUI SEMNAL PWM PENTRU APRINDEREA UNUI LED

Cerință: Să se realizeze o aplicație care să genereze un semnal **PWM software** cu perioada de **1 ms** și factor de umplere de **30%**, aplicat pe un pin al portului **A** pentru controlul unui **LED**. Semnalul trebuie să fie stabil și repetitiv, simulând funcționarea unui modul **hardware** de tip **PWM**.

Sugestii: *Implementarea se va realiza prin folosirea unei bucle infinite în care pinul de ieșire este comutat între nivel logic "**1**" și "**0**", conform timpilor calculați pentru **ON** și **OFF**. Duratele de menținere pe fiecare nivel vor fi obținute prin funcții de întârziere bazate pe cicluri de procesor, proporționale cu frecvența oscilatorului. Factorul de umplere (**duty cycle**) va fi calculat ca procent din perioada totală, astfel încât LED-ul să rămână aprins **30%** din timp și stins **70%**, generând vizual un efect de luminozitate constantă corespunzătoare semnalului **PWM**.*

`main.c`

Fișierul **`main.c`** implementează un program simplu care **aprinde** și **stinge** un **LED** după un ritm prestabilit. Prin calculul unor intervale **ON** / **OFF** se obține un efect de **semnal PWM** cu durată de aprindere proporțională (**duty cycle**). Funcția rulează continuu și poate fi observată atât pe **LED**, cât și pe un pin pentru măsurători externe.

```
/*-----------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul de rulare al aplicației de generare a unui semnal PWM
 *---------------------------------------------------------------------*/

// Includes
#include <ioavr.h>
#include <inavr.h>

/*-----------------------------------------------------------------------
 * Public defines
 *---------------------------------------------------------------------*/

// Frecvența oscilatorului intern = 16000 KHz
#define FOSC 16000UL // KHz
#define TIME 1000UL // ms

// Perioada semnalului
#define PERIOADA TIME * FOSC

// Factorul de umplere
#define DUTY_CYCLE 30

// Timpul în care semnalul corespunde nivelului 1 logic
#define ON_TIME (PERIOADA * DUTY_CYCLE) / 100

// Timpul în care semnalul corespunde nivelului 0 logic
#define OFF_TIME PERIOADA-ON_TIME

// Funcție de __delay_cycles pentru a putea utiliza parametrii definiți
void delay_cycle(unsigned long cycle){
  while(cycle--);
}
```

```
int main( void )
{
  // Se consideră timpul ca fiind măsurat în ms
  unsigned long timp = 1000;
  /*
   * Perioada / Numărul de ciclii
   * Timpul este măsurat în ms, ceea ce înseamnă 100 * 10⁻³s
   * Fosc este măsurată în KHz, ceea ce înseamnă 16000 * 10³ Hz (1/s)
   * Prin urmare, este folosită aceeași unitate de măsură (secunde) peste
   * tot și nu mai trebuie efectuate calcule suplimentare.
   */
  unsigned long nr_ciclii = (timp * (unsigned long) FOSC);

  // Se consideră factorul de umplere (duty-cycle) = 30%, adică T/3
  unsigned long duty_cycle = 30;

  // Perioada în care LED-ul este aprins
  unsigned long onTime = (nr_ciclii * duty_cycle) / 100;

  // Perioada în care LED-ul este stins
  unsigned long offTime = nr_ciclii - onTime;

  // Se setează pinul PA5 ca pin de ieșire
  DDRA |= (1 << PA5);

  // Se setează valoarea pe 1 logic, deci inițial LED-ul este aprins
  PORTA |= (1 << PA5);

  // Se setează pinul PH3 ca pin de ieșire pentru a se vedea pe oscilator
  DDRB |= (1 << PB7);
  PORTB |= (1 << PB7);

  while(1){
    //LED-ul va fi aprins pe perioada onTime
    PORTA |= (1 << PA5);
    PORTB |= (1 << PB7);
    __delay_cycles(ON_TIME);

    //LED-ul va fi stins pe perioada offTime
    PORTA &= ~(1 << PA5);
    PORTB &= ~(1 << PB7);
    __delay_cycles(OFF_TIME);
  }
  return 0;
}
```

6.14.2 CONTROLUL INTENSITĂȚII UNUI LED

Cerință: Să se implementeze un program care să utilizeze **Timer0** în modul **Fast PWM** cu prescaler **8** pentru a controla intensitatea unui **LED** conectat. Intensitatea luminii să varieze gradual, crescând și descrecând periodic între două limite prestabilite. Să se remarce pe osciloscop diferențele dintre formele de undă generate.

Sugestii: *Configurați **TCNT0** în modul **Fast PWM** și setați întreruperile de tip **Overflow** și **Output Compare Match**. Utilizați registrul **OCR0A** pentru a ajusta factorul de umplere **(duty cycle)** și a controla durata de aprindere a LED-ului. Implementați un mecanism de incrementare / decrementare a luminozității **(de tip rampă)** între două valori (**BOTTOM** și **MAX**).*

Pentru aplicația dată, se pun la dispoziție următoarele bibliotecile (headere) și funcțiile aferente:

`timer.h`

Fișierul **`timer.h`** definește funcțiile necesare configurării și utilizării lui **Timer/Counter0**. Acesta conține enum-ul **`TCNT0_Waveform_Mode`** pentru modurile de lucru ale timerului, cât și enum-ul **`TCNT0_Clock_Select`** pentru selecția sursei de **clock** / **prescaler**. Sunt declarate funcții pentru configurarea timerului, precum și pentru activarea / dezactivarea întreruperilor de **overflow** și **Output Compare Match**. De asemenea, include vectorii de întrerupere corespunzători pentru **Timer0**.

```
/*------------------------------------------------------------------------
 * Fișier: timer.h
 * Utilizat pentru declararea funcțiilor pentru Timer
 *---------------------------------------------------------------------*/

#ifndef __PWM__LIB__H__
#define __PWM__LIB__H__

// Includes
#include <ioavr.h>
#include <inavr.h>

/*------------------------------------------------------------------------
 * Type definitions
 *---------------------------------------------------------------------*/

// Definirea modurilor de funcționare pentru Timer/Counter0
typedef enum{
  // WGM2 = WGM1 = WGM0 = 0;
  NORMAL_MODE = 0,

  // WGM2 = WGM1 = 0, WGM0 = 1, TOP = 0XFF
  PHASE_CORRECT = 1,

  // WGM2 = 0, WGM1 = 1, WGM0 = 0
  CTC = 2,

  // WGM2 = 0, WGM1 = WGM0 = 1, TOP = OXFF
  FAST_PWM = 3,

  // WGM2 = 1, WGM1 = 0, WGM0 = 1, TOP = OCRA
  PHASE_CORRECT_OCRA_TOP = 5,

  // WGM2 = WGM1 = WGM0 = 1, TOP = OCRA
  FAST_PWM_OCRA_TOP = 7

} TCNT0_Waveform_Mode;


// Definirea setărilor pentru sursa de ceas (clock) și prescaler
typedef enum{
  // CS02 = CS01 = CS00 = 0, TCNT este oprit
  NO_CLK = 0,

  // CS02 = CS01 = 0, CS00 = 0, fără prescaler
  PRESCALER_1 = 1,

  // CS02 = 0, CS01 = 1,CS00 = 0, prescaler = 8
  PRESCALER_8 = 2,

  // CS02 = 0,CS01 = CS00 = 1, prescaler = 64
  PRESCALER_64 = 3,
```

```
   // CS02 = 1,CS01 = CS00 = 0, prescaler = 256
   PRESCALER_256 = 4,

   // CS02 = 1,CS01 = 0,CS00 = 0, prescaler = 1024
   PRESCALER_1024 = 5,

   // CS02 = CS01 = 1, CS00 = 0, CLK extern conectat la Tn, front negativ
   EXT_CLK_FALLING_EDGE = 6,

   // CS02 = CS01 = CS00 = 1, CLK extern conectat la Tn, front pozitiv
   EXT_CLK_RISING_EDGE = 7

} TCNT0_Clock_Select;

/*-------------------------------------------------------------------------
 * Public (exported) functions
 *-----------------------------------------------------------------------*/

// Funcție utilizată pentru a seta modul pentru TCNT0
void setup(TCNT0_Waveform_Mode mode, TCNT0_Clock_Select sel);

// Vectorul de întrerupere TIMER0_COMPA pentru TCNT0
#pragma vector = TIMER0_COMPA_vect
__interrupt void isr_TIMER0_COMPA();

// Vectorul de întrerupere TIMER0_OVF pentru TCNT0
#pragma vector = TIMER0_OVF_vect
__interrupt void isr_TIMER0_OVF();

// Funcție ce activează întreruperile de overflow
void enable_interrupt_overflow();

// Funcție ce activează întreruperile de Output Compare Match
void enable_interrupt_output_compare();

// Funcție ce dezactivează întreruperile de overflow
void disable_interrupt_overflow();

// Funcție ce dezactivează întreruperile de Output Compare Match
void disable_interrupt_output_compare();

#endif
```

timer.c

Fișierul **timer.c** implementează funcțiile declarate în **timer.h**, oferind codul efectiv pentru setarea modului de funcționare al **TCNT0**, selectarea prescaler-ului, și gestionarea întreruperilor **overflow** și **Output Compare**. Aici se inițializează registrele **TCCR0A/B**, pinii **PA5** și **PB7** și se permite controlul precis al temporizărilor și semnalelor **PWM**.

```
/*-------------------------------------------------------------------------
 * Fișier: timer.c
 * Utilizat pentru definirea funcțiilor declarate în timer.h
 *-----------------------------------------------------------------------*/

// Includes
#include "timer.h"

/*-------------------------------------------------------------------------
 * Public functions (the ones from .h)
 *-----------------------------------------------------------------------*/
```

```
// Funcție utilizată pentru a seta modul pentru TCNT0
void setup(TCNT0_Waveform_Mode mode, TCNT0_Clock_Select sel){
  // PA5 e pinul corespunzator pentru LED
  // Se setează direcția pinului 1 ca ieșire
  DDRA |= (1 << PA5);
  // Inițial, LED-ul va fi stins
  PORTA &= ~(1 << PA5);
  // PB7 e pinul corespunzator TCNT0
  // Se setează pinul ca ieșire
  DDRB |= (1 << PB7);
  // Se setează valoarea pe 0 logic
  PORTB &= ~(1 << PB7);
  // Se resetează registrele
  TCCR0A = 0;
  TCCR0B = 0;

  //Se setează modul de operare
  switch(mode){
  // PWM, Phase Correct, TOP = 0xFF
  case PHASE_CORRECT:
    TCCR0A |= (1 << WGM00) | (1 << COM0A1);
    break;
  // CTC
  case CTC:
    TCCR0A |= (1 << WGM01) | (1 << COM0A0);
    break;
  // Fast PWM, TOP = OXFF
  case FAST_PWM:
    TCCR0A |= (1 << WGM01) | (1<<WGM00)|(1 << COM0A1);
    break;
  // PWM, Phase Correct, TOP = OCRA
  case PHASE_CORRECT_OCRA_TOP:
    TCCR0A |= (1 << WGM00) | (1 << COM0A0);
    TCCR0B |= (1 << WGM02);
    break;
  // Fast PWM, TOP = OCRA
  case FAST_PWM_OCRA_TOP:
    TCCR0B |= (1 << WGM02);
    TCCR0A |= (1 << WGM01) | (1 << WGM00) | (1 << COM0A0);
    break;
  // Default
  default:
    TCCR0A =0;
    TCCR0B=0;
    break;
  }

  // Selecția clock-ului
  switch(sel){

  case PRESCALER_1: TCCR0B |= (1 << CS00);
  break;

  case PRESCALER_8: TCCR0B |= (1 << CS01);
  break;

  case PRESCALER_64: TCCR0B |= (1 << CS00) | (1<<CS01);
  break;

  case PRESCALER_256: TCCR0B |= (1 << CS02);
  break;
```

```
  case PRESCALER_1024: TCCR0B |= (1 << CS00) | (1<<CS02);
  break;

  case EXT_CLK_FALLING_EDGE:TCCR0B |= (1 << CS01) | (1<<CS02);
  break;

  case EXT_CLK_RISING_EDGE:
  TCCR0B |= (1<<CS00) | (1 << CS01) | (1 << CS02);
  break;

  // În cazul default, se setează ca selecția să fie fără prescaler
  default: TCCR0B |= (1 << CS00);
  break;
  }
}

// Funcție ce activează întreruperile de overflow
void enable_interrupt_overflow(){
  TIMSK0 |= (1 << TOIE0);
  //__enable_interrupt();
}

// Funcție ce dezactivează întreruperile de Output Compare Match
void enable_interrupt_output_compare(){
  TIMSK0 |= (1 << OCIE0A);
  //__enable_interrupt();
}

// Funcție ce dezactivează întreruperile de overflow
void disable_interrupt_overflow(){
  TIMSK0 &= ~((1 << TOIE0) | (1 << OCIE0A));
  //__disable_interrupt();
}

// Funcție ce dezactivează întreruperile de Output Compare Match
void disable_interrupt_output_compare(){
  TIMSK0 &= ~(1 << OCIE0A);
  //__disable_interrupt();
}
```

main.c

Codul folosește **Timer0** în mod **Fast PWM** cu prescaler **8** pentru a controla intensitatea unui **LED**. Valoarea registrului **OCR0A** este modificată ciclic între limitele **BOTTOM** și **MAX**, ceea ce schimbă factorul de umplere (**duty cycle**) al semnalului **PWM**. Astfel, **LED-ul** își variază luminozitatea gradual, simulând un efect de **pulsare**.

```
/*-----------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul principal al aplicației de control al intensității unui LED
 *---------------------------------------------------------------------*/

// Includes
#include "timer.h"

/*-----------------------------------------------------------------------
 * Private defines
 *---------------------------------------------------------------------*/
#define BOTTOM  25
#define MAX     230
```

```
// Rutina de întrerupere pentru comparație
__interrupt void isr_TIMER0_COMPA() {
  /*
   * Atunci când se produce o comparație și valorile lui OCRnx și TCNTn
   * sunt egale, LED-ul se va stinge
   */
  PORTA &= ~(1 << PA5);
}

// Rutina de întrerupere pentru overflow
__interrupt void isr_TIMER0_OVF(){
  /*
   * Atunci când se ajunge la valoarea maximă, se activează flag-ul de
   * overflow, LED-ul se va aprinde
   */
  PORTA |= (1 << PA5);
}

// Intensitatea LED-ului
unsigned int brightness = 0;

int main(void) {

  setup(FAST_PWM, PRESCALER_8);
  enable_interrupt_overflow();
  enable_interrupt_output_compare();
  __enable_interrupt();

  // Se setează un pas pentru a scade / crește intensitatea LED-ului
  int pas = 1;

  while (1) {
    brightness += pas;
    // Se pune valoarea respectivă în registrul de comparare al TCNT0
    OCR0A = brightness;
    // Brightness poate lua valoarea maximă de 255, TCNT fiind pe 8 biți
    if (brightness >= MAX) {
      // În cazul în care am atins valoarea maximă, începem să scadem
      brightness = MAX;
      // Se setează pasul la valoarea -1 pentru a putea decrementa
      pas = -1;
    } else if (brightness == BOTTOM) {
      /*
       * În momentul în care brightness ajunge inapoi la valoarea BOTTOM,
       * setăm pasul pe 1 pentru a putea incrementa
       */
      pas = 1;
    }
    __delay_cycles(80000);
  }

  return 0;
}
```

6.14.2.1 REZULTATELE CAPTURATE PE OSCILOSCOP

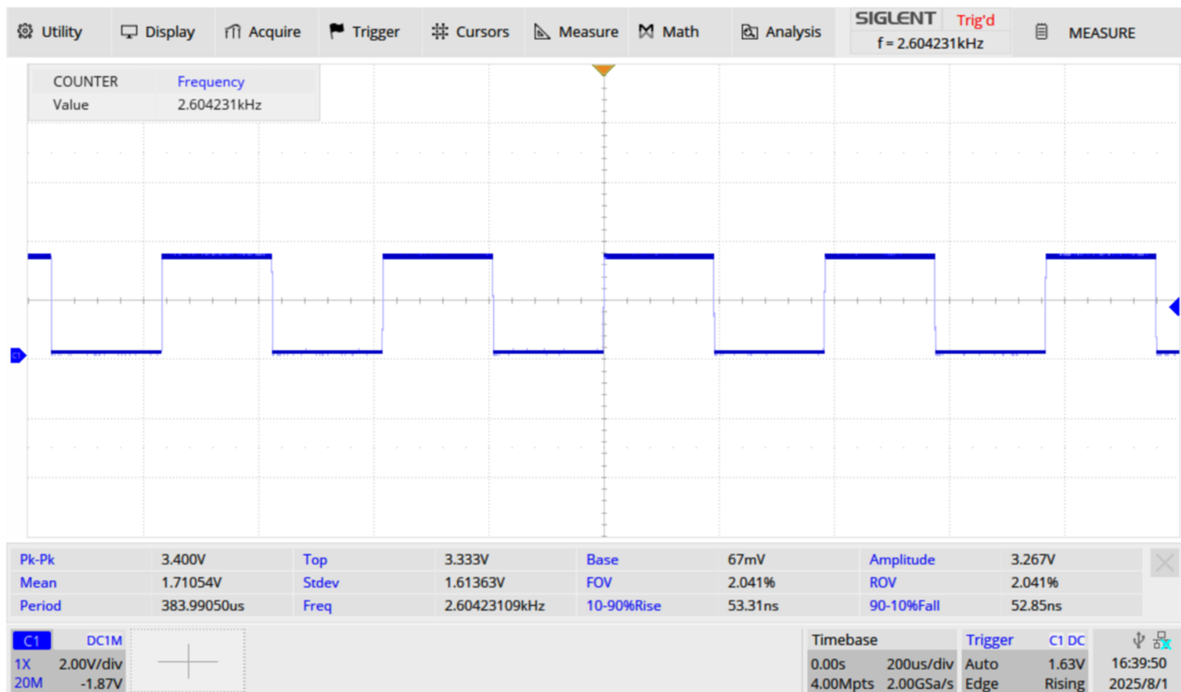


Figura 6.33 – Captura de osciloscop pentru modul Fast PWM

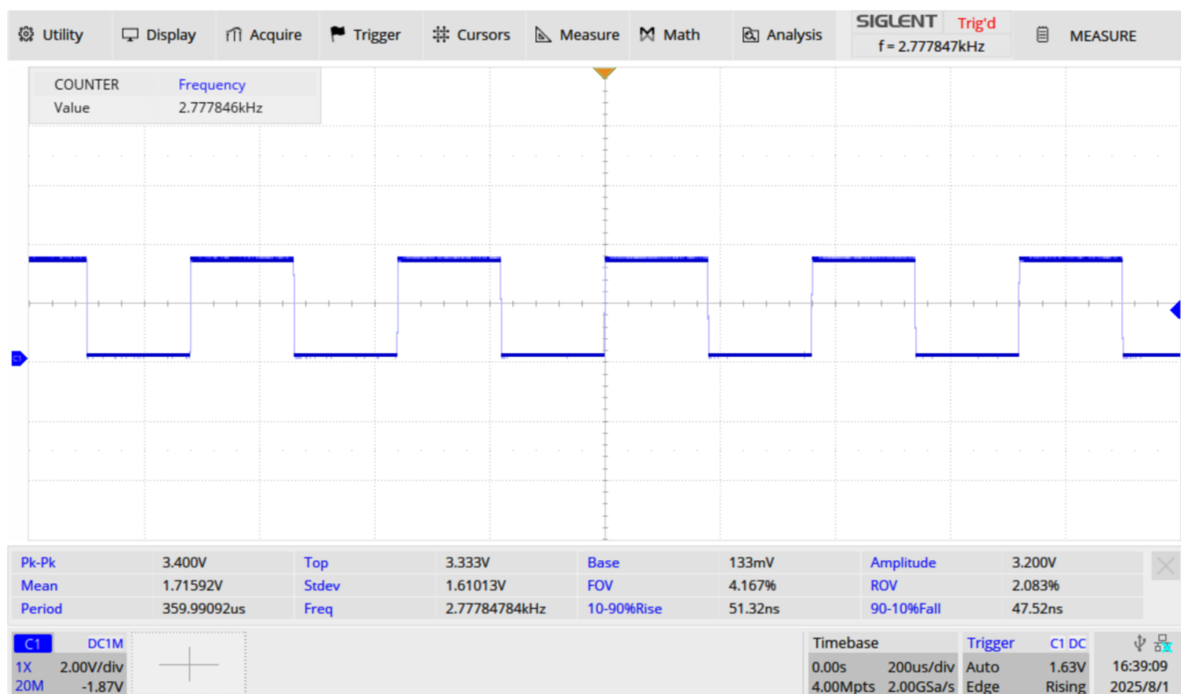


Figura 6.34 – Captura de osciloscop pentru modul Fast PWM, cu counter-ul numărând până la valoarea lui OCRA

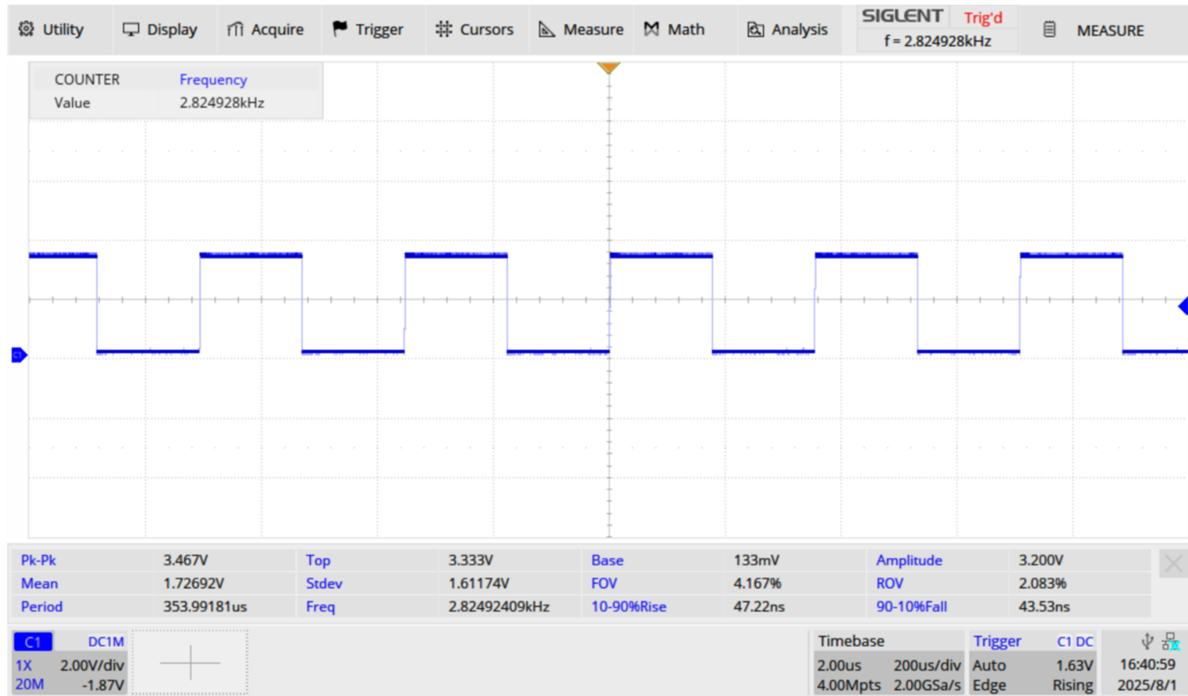


Figura 6.35 – Captura de osciloscop pentru modul Phase Correct

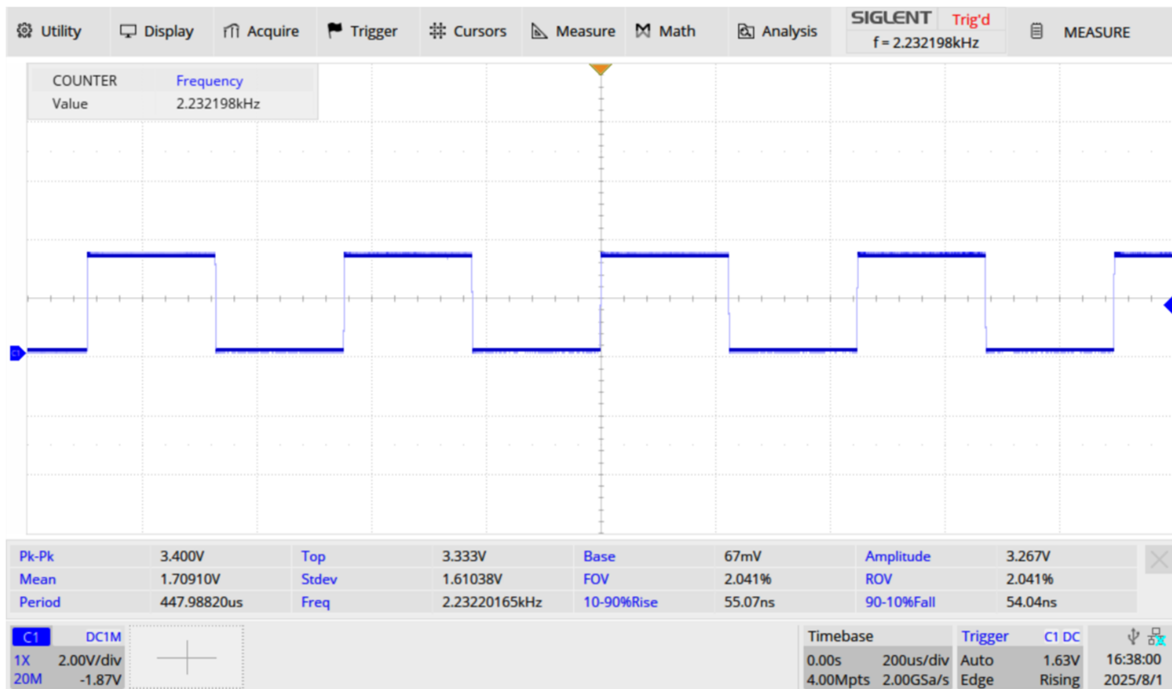


Figura 6.36 – Captura de osciloscop pentru modul Phase Correct, cu counter-ul numărând până la valoarea lui OCRA

6.14.3 IMPLEMENTAREA UNUI PERIODMETRU (VARIANTA BLOCANTĂ)

Cerință: Să se realizeze un program care să măsoare **perioada** și **frecvența** unui semnal aplicat pe un pin digital, folosind **Timer/Counter1** în modul **blocant**. Aplicația trebuie să calculeze perioada în **microsecunde** (**µs**) și frecvența semnalului în **hertzi** (**Hz**) și să le stocheze în variabile vizualizabile în debugger.

Sugestii: *Se poate folosi frontul pozitiv al semnalului pentru sincronizare inițială, urmat de măsurarea numărului de ciclii ai timerului între două fronturi consecutive. Prescaler-ul temporizatorului trebuie ales astfel încât să permită măsurarea exactă a semnalului* ***fără overflow****. Pentru validare, se pot utiliza* ***breakpoint-uri*** *și* ***Watch*** *pentru a vizualiza valorile calculate.*

`main.c`

Fișierul **`main.c`** este punctul de intrare al aplicației, unde se configurează pinul de intrare pentru semnalul de măsurat și **Timer/Counter1**. Programul sincronizează măsurarea cu primul front pozitiv, măsoară numărul de ciclii pentru o perioadă completă a semnalului, calculează perioada și frecvența și actualizează continuu valorile în variabile vizualizabile. Aplicarea este **blocantă**, iar toate calculele și măsurătorile se fac secvențial.

```
/*-----------------------------------------------------------------------
 * Fișier: main.c
 * Utilizat la implementarea unui periodmetrului (varianta blocantă)
 *---------------------------------------------------------------------*/

// Includes
#include <ioavr.h>
#include <inavr.h>

// Frecvența 16 MHz
#define F_CPU 16000000UL

// Fără prescaler
#define PRESCALER 8UL

// Perioada măsurată în microsecunde (µs)
volatile unsigned long perioada_us;

// Frecvența
volatile unsigned long frecventa;

int main( void )
{
  /*
   * Se declară acest pas pentru a putea utiliza breakpoint-ul pentru a
   * vizualiza valorile din Watch
   */
  int i = 0;

  /*
   * Se setează pinul PF3 ca pin de intrare prin intermediul căruia
   * primim semnalul generat
   */
  DDRF &= ~(1 << PF3);

  // Numărul de ciclii
  unsigned long nr_cicli;

  while(1){
    // TCNT1 este oprit
    TCCR1A = 0;
    TCCR1B = 0;
```

```
    // Primul front e "sacrificat" pentru sincronizare
    while ((PINF & (1 << PF3)) == 0);
    while ((PINF & (1 << PF3)) != 0);

    // Se resetează valoarea contorului
    TCNT1 = 0;

    // Fără prescaler
    TCCR1B |= (1 << CS11);

    // Se așteaptă frontul pozitiv
    while ((PINF & (1 << PF3)) == 0);

    // Se așteaptă apoi cel negativ
    while ((PINF & (1 << PF3)) != 0);

    //Se oprește temporizatorul
    TCCR1B = 0;

    //Se salvează valoarea contorului
    nr_cicli = TCNT1;

    //Se calculează perioada în ms
    perioada_us = (nr_cicli * PRESCALER) / (F_CPU / 1000000UL);
    frecventa = 1000000UL / perioada_us;

    i++; // Aici se pune breakpoint pentru a vedea valorile în Watch
  }
  return 0;
}
```

6.14.4 IMPLEMENTAREA UNUI PERIODMETRU (VARIANTA NEBLOCANTĂ)

Cerință: Să se realizeze un **periodmetru** în varianta **neblocantă**, care să măsoare **perioada** și **frecvența** unui semnal aplicat pe un pin. Comparați funcționarea cu varianta **blocantă** și argumentați care este mai eficientă.

Sugestii: *Se poate folosi un temporizator intern (**TCNT1**) și întreruperi pe schimbarea nivelului pinului pentru a detecta fronturile semnalului. Calculul perioadei și frecvenței se face doar atunci când semnalul a fost **măsurat complet**, lăsând microcontrolerul liber să execute alte sarcini între evenimente. Această abordare evită **blocarea** procesorului în așteptarea fronturilor semnalului.*

`main.c`

Fișierul **`main.c`** implementează un **periodmetru neblocant** utilizând un temporizator intern și întreruperi pentru a măsura intervalul dintre fronturile unui semnal. Perioada și frecvența sunt calculate doar după ce s-a detectat un ciclu **complet**, în timp ce microcontrolerul poate continua să execute alte operații între fronturi. Logica de control asigură că temporizatorul este pornit și oprit automat la fronturi, iar întreruperile permit o măsurare precisă **fără a bloca** bucla principală.

```
/*-----------------------------------------------------------------------
 * Fișier: main.c
 * Utilizat la implementarea unui periodmetrului (varianta neblocantă)
 *-------------------------------------------------------------------*/

// Includes
#include <inavr.h>
#include <ioavr.h>

// Frecvența CPU = 16 MHz
#define F_CPU 16000000UL
```

```
// Factorul de umplere (duty-cycle) = clkI/O/8
#define PRESCALER 8UL

// Starea temporizatorului
static unsigned char stare = 0;

// Indică dacă se poate sau nu măsura perioada / frecvența
static unsigned char readState = 0;

// Perioada în microsecunde care trebuie măsurată
volatile unsigned long perioada_us;

// Frecvența în Hz care trebuie măsurată
volatile unsigned long frecventa;

// Numărul de ciclii parcurși de temporizator
unsigned long nr_ciclii;

/*
 * Vectorul de întrerupere pentru PCINT7:0
 * ATENȚIE: Nu există vectorul de întrerupere PCINT5 în tabela TVI
 */
#pragma vector = PCINT0_vect
__interrupt void isr_PCINT0(void){
  // Se așteaptă un front pozitiv
  if((PINB & (1 << PB5)) != 0){
    return;
  }

  switch (stare){
  // În starea S = 0 temporizatorul este pornit
  case 0:
    TCCR1B |= (1 << CS11);
    TCNT1 = 0;
    stare = 1;
    break;

// În S = 1 temporizatorul e oprit și se citește nr. de ciclii parcurși
  case 1:
    TCCR1B &= ~(1 << CS11);
    nr_ciclii = TCNT1;
    readState = 1;
    break;

  default:
    break;
  }
}

int main( void )
{
  // Se setează pinul PB5 ca pin de intrare
  DDRB &= ~(1<<PB5);

  // Timer-ul inițial este oprit
  TCCR1A = 0;
  TCCR1B = 0;

  // PCICR - Pin Change Interrupt Control Register
  PCICR |= (1 << PCIE0);
```

```
  /*
   * PCMSK0 - Pin Change Mask Register 0
   * Se activează întreruperea pentru PCINT5
   */
  PCMSK0 |= (1 << PCINT5);

  __enable_interrupt();

  while(1){

    switch (readState){

    case 0:
      // Nu se întâmplă nimic aici
      break;

    case 1:
      /*
       * Atunci când readState ia valoarea 1, putem măsura frecvența și
       * perioada
       * Se dezactivează întreruperile pentru o măsurare corectă
       */
      __disable_interrupt();
      PCMSK0 &= ~(1 << PCINT5);

      // Se calculează perioada și frecvența în funcție de nr. de ciclii
      perioada_us = (nr_ciclii * PRESCALER) / (F_CPU / 1000000UL);
      frecventa = 1000000UL / perioada_us;

      // Se revine în starea S = 0
      Stare = 0;
      readState = 0;

      // Se activează întreruperile
      PCMSK0 |= (1 << PCINT5);
      __enable_interrupt();
      break;

    default:
      break;
    }
  }
  return 0;
}
```

6.14.5 GENERAREA UNUI SEMNAL MODULAT

Cerință: Să se configureze **Timer0** și **Timer1** pentru generarea unui semnal **modulat** și să se vizualizeze semnalele pe **osciloscop**. Determinați pe ce canal se află semnalul modulant și pe ce canal semnalul purtător.

Sugestii: *Se pot folosi modurile **Fast PWM** ale celor două temporizatoare, activând canalele corespunzătoare pentru ieșire. Setarea prescaler-ului și a valorii **OCR** permite ajustarea **frecvenței** și a **duty-cycle-ului** semnalelor. Vizualizarea pe osciloscop ajută la identificarea canalelor semnalului modulant și purtător.*

main.c

Fișierul **main.c** configurează **Timer0** și **Timer1** în modul **Fast PWM** pentru a genera semnale pe mai multe canale (**A**, **B**, **C**). **Timer0** controlează semnalele pe **PB7** și **PB5**, iar **Timer1** pe **PG5** și **PB5**, fiecare canal având **duty-cycle** setat și **prescaler** corespunzător. Structura codului permite modularea semnalului prin ajustarea canalelor și observarea rezultatelor pe osciloscop, separând semnalul purtător de semnalul modulant.

```c
/*---------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul principal al aplicației de generare a unui semnal modulat
 *-------------------------------------------------------------------*/

// Includes
#include <ioavr.h>
#include <inavr.h>

int main( void )
{
  // Pinii de ieșire
  DDRB  |= (1 << PB7) | (1 << PB5);
  DDRG  |= (1 << PG5);

  /*
   * TCNT0 - canal A și B
   * Se resetează regiștrii TCNT0
   */
  TCCR0A = 0;
  TCCR0B = 0;

  // Se setează pe activ canalul A și modul de operare Fast PWM
  TCCR0A |= (1 << COM0A1) | (1 << WGM01) | (1 << WGM00);

  // Se setează pe activ canalul B
  TCCR0A |= (1 << COM0B1);

  // Se selectează un prescaler clkIO/256
  TCCR0B |= (1 << CS02);

  // Duty-cycle de 50% pe ambele canale
  OCR0A = 127;
  OCR0B = 127;

  /*
   * TCNT0 - canal A și B
   * Se resetează regiștrii TCNT0
   */
  TCCR1A = 0;
  TCCR1B = 0;

  // Se setează TCNT1 în modul Fast PWM 8 bit și se activează canalul C
  TCCR1A |= (1 << WGM10) | (1 << COM1C1);

  // Se activează canalul A
  TCCR1A |= (1 << COM1A1);

  // Se selectează prescaler clkIO/8
  TCCR1B |= (1 << CS11) | (1 << WGM12);

  return 0;
}
```

Figura 6.37 – Captura de osciloscop a semnalului modulat

6.14.6 CONFIGURAREA LUI TIMER2 ÎN MODUL ASINCRON

Cerință: Se cere realizarea unei aplicații în care **Timer2** funcționează în modul **asincron** folosind un oscilator extern, cu comutarea unui **LED** și a unui pin de ieșire la fiecare **overflow**. Trebuie observată diferența între modul **sincron** și **asincron**, precum și determinată **frecvența oscilatorului** de cristal extern.

Sugestii: *Se recomandă configurarea lui* ***TCNT2*** *pentru modul asincron prin setarea bitului* ***AS2****, selectarea prescaler-ului adecvat și folosirea* ***ISR*** *pentru toggle-ul* ***LED-ului*** *și pinului de semnalizare. Așteptarea actualizării registrelor asincrone este necesară pentru stabilitatea temporizatorului.*

main.c

Fișierul **main.c** configurează **Timer2** în modul **asincron**, folosind un **cristal extern** ca sursă de timp. Overflow-ul temporizatorului declanșează întreruperi care **comută** un LED și un pin pentru vizualizare pe osciloscop. Această implementare permite măsurători de timp mai precise și independente de frecvența internă a microcontrolerului, evidențiind diferențele între operarea sincronă și asincronă.

```
/*------------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul utilizat pentru configurarea lui Timer2 în modul asincron
 *----------------------------------------------------------------------*/

// Includes
#include <ioavr.h>
#include <inavr.h>

// Rutina de întrerupere pentru overflow
#pragma vector = TIMER2_OVF_vect
__interrupt void TIMER2_OVF_ISR(void)
{
    PORTA ^= (1 << PA5);
    PORTB ^= (1 << PB7);
}
```

```
int main( void )
{
  // Se setează pinul PA5 ca pin de ieșire pentru LED A
  DDRA |= (1 << PA5);
  // Se setează pinul PB7 ca ieșire pentru vizualizarea pe osciloscop
  DDRB |=(1 << PB7);
  // Se selectează sursa de clock externă
  ASSR |= (1 << AS2);
  // Valoarea inițială a numărătorului
  TCNT2 = 0;
  // Se setează prescaler-ul la 128
  TCCR2B |= (1 << CS22) | (1 << CS20);

  // Se așteaptă actualizarea unuia dintre aceste registre
  while (ASSR & ((1 << TCR2BUB) | (1 << TCR2AUB) | (1 << OCR2BUB)
               | (1 << OCR2AUB) | (1 << TCN2UB)));

  // Se activează întreruperea de overflow
  TIMSK2 |= (1 << TOIE2);

  __enable_interrupt();

  while(1);

  return 0;
}
```

6.15 BIBLIOGRAFIE

1. *"8-bit AVR Microcontroller ATmega 1280 Datasheet", Microchip Technology*
2. *"Schematic for UNI Clicker", Mikroe*
3. *"Schematic for ATmega1280: SiBrain", Mikroe*
4. *"Timer / Counter", Wikipedia*
5. *"Pulse Width Modulation", Wikipedia*
6. *"Timers And Counters", Geeks For Geeks*

7. RESET & WATCHDOG

7.1 UNDE SE FOLOSEȘTE RESET & WATCHDOG?

Watchdog Timer (**WDT**) este utilizat pe scară largă în sisteme electronice și embedded pentru a crește fiabilitatea și a preveni blocarea aplicațiilor. Principalele domenii de utilizare includ:

- **Automotive:** monitorizarea unităților de control electronice pentru a detectarea erorilor software critice;
- **Sisteme industriale:** protecția sistemelor de control împotriva blocajelor;
- **Electronice de consum:** mentenanța dispozitivelor precum router-elor, TV-urilor și a electrocasnicelor;
- **Aerospațial și militar:** garantarea funcționării continue a echipamentelor în medii critice;
- **Dispozitive medicale:** asigurarea siguranței pacienților prin repornirea automată a aparatelor.

7.2 CUNOȘTINȚE NECESARE

Pentru a putea parcurge acest capitol, este necesar să cunoașteți următoarele subiecte din capitolele anterioare, și nu numai:

- Utilizarea întreruperilor;
- Utilizarea timerelor pentru întreruperi sincronizate și generarea semnalelor **PWM**;
- Modul de funcționare al memoriilor **EEPROM**;
- Utilizarea kit-ului hardware Mikroe.

7.3 ABSTRACT

Acest capitol are în vedere descrierea resetării sistemului și cauzele acesteia, punând în evidență mecanismul Watchdog-ului.

7.4 HARDWARE ȘI SOFTWARE NECESAR

- ATMega1280 SiBRAIN;
- UNI Clicker;
- EEPROM Click;
- Atmel ICE;
- IAR *Embedded Workbench* IDE 7.30.5;
- Osciloscop.

7.5 INTRODUCERE ÎN CONTROLUL ȘI RESETAREA SISTEMULUI

În timpul resetării, toți regiștrii de 1/0 sunt readuși la valorile lor inițiale determinate de hardware, iar programul își începe execuția de la adresa vectorului Reset(adresa fixa 0x0000). Instrucțiunea plasată în vectorul de Reset ar trebui să fie o instrucțiune de salt, și anume o instrucțiune de salt absolut(JMP) spre rutina care gestionează resetarea.

Întreruperile sunt dezactivate implicit după reset, prin dezactivarea bitului global de întreruperi, pentru a preveni întreruperi premature care ar putea perturba procesul de inițializare. Vectorul de întreruperi este folosit doar după ce întreruperile sunt activate explicit prin setarea bitului global. Dacă întreruperile nu sunt activate, execuția programului continuă normal, secvențial, fără să fie influențată de întreruperi. Astfel, vectorul de reset asigură un punct clar și sigur de pornire a programului, iar vectorii de întreruperi sunt utilizați numai când întreruperile sunt permise și active.

7.6 SURSELE DE RESETARE

Microcontrolerul Atmega1280 are 5 surse de resetare:

- **Resetarea la pornire(Power-on Reset):** microcontrolerul este resetat, atunci când tensiunea de alimentare crește de la zero și se află sub pragul de resetare la pornire; procesorul începe execuția abia după ce V_{CC} este stabilă.
- **Resetarea externă(External Reset):** atunci când pinul RESET este menținut la nivel logic 0 pentru o durată mai mare decât perioada minimă necesară pentru semnalul de reset.
- **Resetarea prin Watchdog(Watchdog Reset): d**acă temporizatorul watchdog nu este resetat la timp de către program, el generează un reset automat pentru a evita blocarea sistemului.
- **Resetarea de tip brown-out(Brown-out Reset):** fenomenul de **brown-out** constă în scăderea valorii tensiunii într-un circuit electric sub un anumit prag, numit prag de **brown-out**; microcontrolerul este resetat când valoarea tensiunii de alimentare $\mathbf{V_{CC}}$ este inferioară valorii pragului de resetare **brown-out** ($\mathbf{V_{BOT}}$) și detectorul fenomenului de **brown-out** este activat.
- **Resetarea de tip JTAG AVR:** microcontrolerul e resetat când registrul de **Reset** are valoarea **1** logic.

7.7 INTRODUCERE ÎN WATCHDOG

Ideea de bază ce stă în spatele existenţei unui **watchdog timer** este de a avea un mecanism care să reseteze microcontrolerul în cazul în care dintr-un motiv excepțional, aplicaţia nu răspunde un timp îndelungat. Această funcție este esențială pentru aplicații în timp real și sisteme embedded sigure și fiabile.

Familia ATMega AVR oferă un **watchdog** intern foarte robust cu o sursă de **clock** separată faţă de microcontroler. O eroare a **clock-ului** principal nu afectează **watchdog-ul.**

Clarificarea unor termeni folosiţi în acest laborator:

- **Watchdog Timer** (**WDT**) - modul periferic care poate fi configurat să genereze un semnal de reset, dacă este resetat prea devreme sau prea târziu potrivit unei perioade specificate;
- **Watchdog Timer Reset** (**WDT Reset**) – resetarea registrului **WDT** (revenirea la o valoare stabilită inițial;
- **Resetarea sistemului** – resetarea microcontrolereului AVR.

Prezintă următoarele aspecte:

- Semnalul de ceas este separat și provine de la un oscilator **on-chip**.
- Are 3 moduri de operare: **Interrupt**, **System Reset** și **Interrupt and System Reset**.
- O perioadă de time-out selectabilă de la **16 ms** la **8 s**.

7.7.1 SCHEMA DE PRINCIPIU

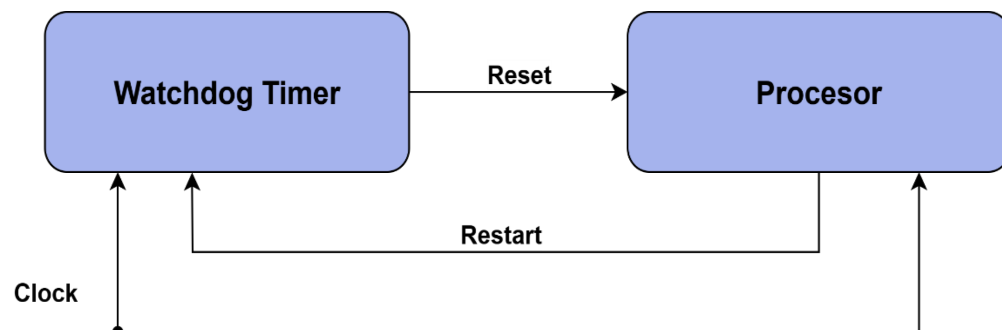


Figura 7.1 – Schema de principiu

Definiție

Watchdog-ul este un timer care contorizează ciclurile semnalului de ceas pe un oscilator on-chip separat, ce are frecvența de 128 kHz.

Acesta produce o întrerupere sau o resetare a sistemului atunci când contorul său atinge valoarea de expirare a timpului (**time-out**). În modul normal de operare, pentru a preveni resetarea, programul trebuie să execute periodic instrucțiunea specială WDR (Watchdog Timer Reset), care resetează contorul intern al WDT-ului, împiedicând astfel atingerea time-out-ului. Dacă sistemul nu îl resetează, o întrerupere sau o resetare a sistemului va fi produsă.

La nivel intern, acesta trebuie resetat de către software la intervale de timp regulate pentru ca numărătorul acestuia să nu atingă valoarea **0**, caz în care se produc întreruperi și / sau reset-uri nedorite.

Astfel, Watchdog Timer-ul servește ca o „paznic” ce resetează automat sistemul dacă software-ul nu confirmă în mod regulat că rulează corect, prevenind astfel blocările sau situațiile în care sistemul nu mai răspunde.

7.7.2 SINCRONIZAREA ÎN DOMENII DIFERITE DE CLOCK

WDT şi **UCP** operează în domenii de **clock** diferite, iar sincronizarea între aceste domenii trebuie luată în considerare atunci când folosim **watchdog-ul**. Pentru a configura **WDT-ul** sunt necesare **2-3 perioade de clock** ale **WDT-ului**. Setările sunt scrise în registrele de control ale **WDT-ului**, acestea fiind efective la următorul front pozitiv al **clock-ului watchdog-ului**, adică după **2-3 ms** după ce setările au fost scrise în registru. De aici rezultă că perioada iniţială de time-out este cu **3 ms mai lungă**. Dacă perioada de time-out este de **8 ms**, perioada actuală este între **10 şi 11 ms**. Acest lucru este relevant atunci când se foloseşte modul fereastră şi perioade de **time-out** mici. Această caracteristică nu este specifică doar **watchdog-ului**; toate timer-ele asincrone operează în acest fel pentru sincronizarea clock-ului între diferite domenii. **WDT-ul** este resetat atunci când are loc o scriere validă în registrele de control.

O altă caracteristică de sincronizare este diferenţa de timp dintre executarea instrucţiunii de **WDT reset** şi resetarea acestuia efectiv, problemă ce ţine de sincronizare dintre domenii de **clock**. **WDT** este resetat după 3 perioade de **clock** după ce instrucţiunea de reset este executată, adică între **2-3 ms** după executarea instrucţiunii. Dacă se foloseşte o perioadă de time-out de **8 ms**, prima instrucţiune de **WDT** reset va fi executată după **5 ms** de la activarea **watchdog-ului**. Ţinând cont de precizia oscilatorului ± 30%, instrucţiunea trebuie să fie executată în **3.5 ms** sau mai puţin. Intervalul dintre două instrucţiuni de **WDT reset** poate fi de **4.9 ms sau mai mic**:

T = 8 ms – 1 ms incertitudine - 30% precizia oscilatorului

Efectul acestei sincronizări este minimizat atunci când perioada de time-out este **mai mare**. Dacă **WDT** generează un semnal de reset (de exemplu după expirarea perioadei de time-out) resetarea sistemului are loc la următorul front al **clock-ului watchdog-ului**. Resetarea sistemului are loc la **1 ms** după ce perioada de time-out a expirat. Acest lucru în mod normal nu ar trebui să creeze nici o problemă, dar este important de ştiut dacă se doreşte să se măsoare perioada watchdog-ului urmărind nivelului logic de tensiune al unui pin. Cea mai bună metodă de calcul a perioadei efective a watchdog-ului este folosirea **MEGA Real Time Clock Timer-ului**.

În modul normal, Watchdog trebuie resetat înainte ca temporizatorul să ajungă la timpul de expirare (time-out) pentru a preveni resetarea sistemului.
În modul fereastră, Watchdog poate fi resetat numai într-un interval de timp specific (fereastra). Resetarea în afara acestui interval, prea devreme sau prea târziu, va cauza resetarea procesorului. Acest mod oferă un control mai strict asupra momentului în care este făcut resetul pentru a detecta erori mai complexe.

Toate aceste caracteristici sunt valabile atât în modul normal cât şi în modul fereastră.

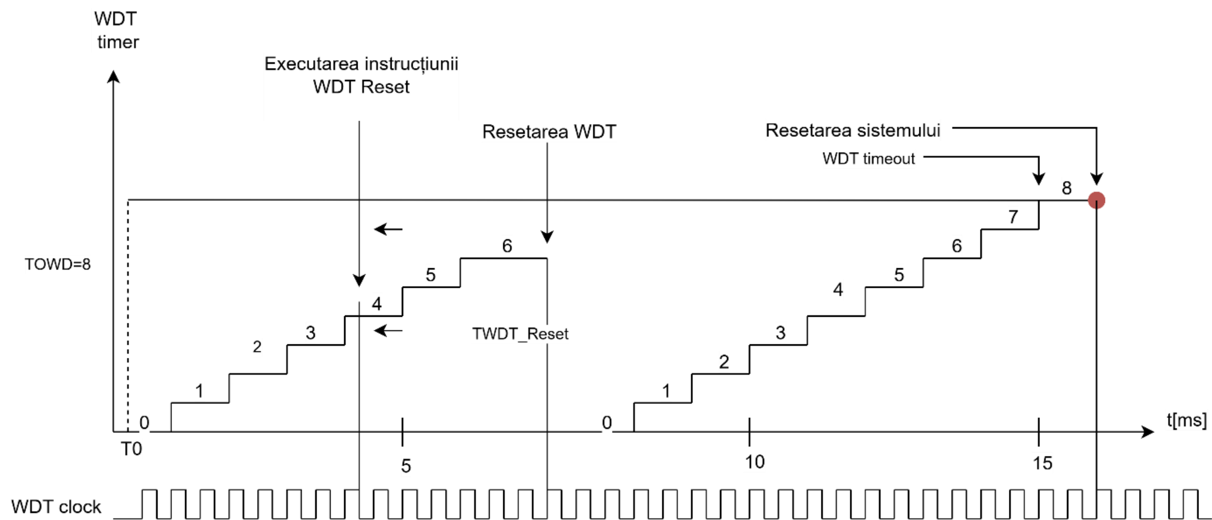


Figura 7.2 – Reprezentarea grafică a ciclurilor Watchdog Timer (WDT) de la inițializare (T0) până la resetarea manuală (TWDT_Reset)

7.7.3 MODURILE DE FUNCȚIONARE

7.7.3.1 INTERRUPT MODE (MODUL DE ÎNTRERUPERI)

Watchdog-ul produce o **întrerupere** atunci când timpul setat **expiră**, aceasta fiind folosită pentru a "trezi" anumite dispositive din modul sleep. Un exemplu de utilizare a acestui mod este de a limita timpul maxim al unei anumite operații, producându-se o întrerupere atunci când acesta este atins. Astfel, operația nu rulează mai mult timp decât este așteptat.

7.7.3.2 SYSTEM RESET MODE (MODUL DE RESETARE AL SISTEMULUI)

Watchdog-ul produce o **resetare** atunci când timpul **expiră**. Acesta este utilizat pentru a preveni blocarea sistemului în cazul în care se execută cod interminabil.

7.7.3.3 INTERRUPT AND SYSTEM RESET MODE

Interrupt and System Reset Mode combină celelalte 2 moduri de funcționare. În primul rând se produce **întreruperea**, apoi **resetarea** sistemului. Dacă bitul **WDE** este setat, **watchdog-ul** se află în modul acesta. Primul time-out al numărătorului va seta **WDIF**, iar executarea vectorului de întreruperi va reseta biții **WDIE** și **WDIF** cu ajutorul hardware-ului, watchdog-ul trecând în modul de resetare. Pentru a-l menține în modul de întreruperi, bitul **WDIE** trebuie setat după fiecare întrerupere. Acest lucru nu ar trebui realizat folosind rutina de întrerupere pentru că s-ar putea compromite funcționalitatea de siguranță a modului de reset. Dacă întreruperea nu se exexcută înainte de time-out, se va produce o resetare a sistemului. **Interrupt and System Reset Mode** este folosit pentru închiderea în manieră sigură a dispozitivului, salvând date critice (parametri critci) înainte de un reset.

7.8 STANDARDUL INTERNAȚIONAL IEC 60730

Definiție

IEC 60730 este un standard de siguranță pentru aparatele de uz casnic, care abordează aspectele de design și funcționare a produselor.

Acest standard este menționat și de alte standarde care vizează siguranța dispozitivelor, de exemplu standardul **IEC 60335**. Pentru siguranță este foarte important ca sistemul să fie certificat de acest standard.

7.8.1 CLASELE STANDARDULUI IEC 60730

Anexa H a standardului definește **3 clase de control** ale software-ului pentru diferite electrocasnice:
- **Clasa A** – funcții de control care nu sunt destinate a fi invocate pentru siguranța echipamentelor;
- **Clasa B** – aplicații care includ cod cu scopul de a preveni alte erori decât cele **software**;
- **Clasa C** – aplicații cu cod destinat prevenirii erorilor fără utilizarea dispozitivelor de protecție.

7.8.2 WATCHDOG-UL DE CLASĂ B

Arhitectura **MEGA** este prevăzută cu un mecanism de protecție care asigură faptul că setările **WDT** nu pot fi modificate accidental. Pentru o siguranță sporită este prevăzută o siguranță (**fuse**) pentru a bloca setările **WDT**.

Deoarece **WDT** este un element de siguranță integrat în familia **Atmel AVR MEGA**, s-a conceput o rutină de autodiagnosticare care testează ambele moduri de funcționare, normal și fereastră. Aceasta se execută după resetare, în partea de pre-inițializare a aplicației înaintea funcției main.

Rutina de autodiagnosticare ne asigură că:
- **Resetarea timer-ului** este realizată după expirarea perioadei de time-out a watchdog-ului.
- **Watchdog timer-ul** poate fi resetat.
- **Sistemul este resetat** la resetarea prematură a watchdog-ului în modul de funcționare fereastră.

Conform diagramei logice, dispozitivul este resetat de câteva ori în timpul testului. Prin urmare variabila **SRAM** și flag-urile de reset ale dispozitivului sunt utilizate de rutina de autodiagnosticare, pentru a urmări faza de testare. În continuare utilizatorul poate configura ce să facă în cazul unui reset **software**, **brown-out** sau cum să proceseze resetul cauzat de watchdog, atunci când testul se află în starea **WDT_OK**.

Rutina de autodiagnosticare folosește un **Real Time Counter** (**RTC**) pentru a verifica perioada Watchdog-ului. **RTC-ul** are o sursă de clock independentă față de **CPU** și **WDT**. Ambele module au oscilatoare care funcționează la **32.768 kHz**. Cu toate acestea oscilatorul **WDT-ului** este optimizat pentru un consum redus de energie. **RTC-ul** este folosit pentru estimarea perioadei **WDT-ului**, iar programul verifică dacă această perioadă este în intervalul (**T / 2, 3T / 2**), unde **T** este **perioada nominală a WDT-ului**.

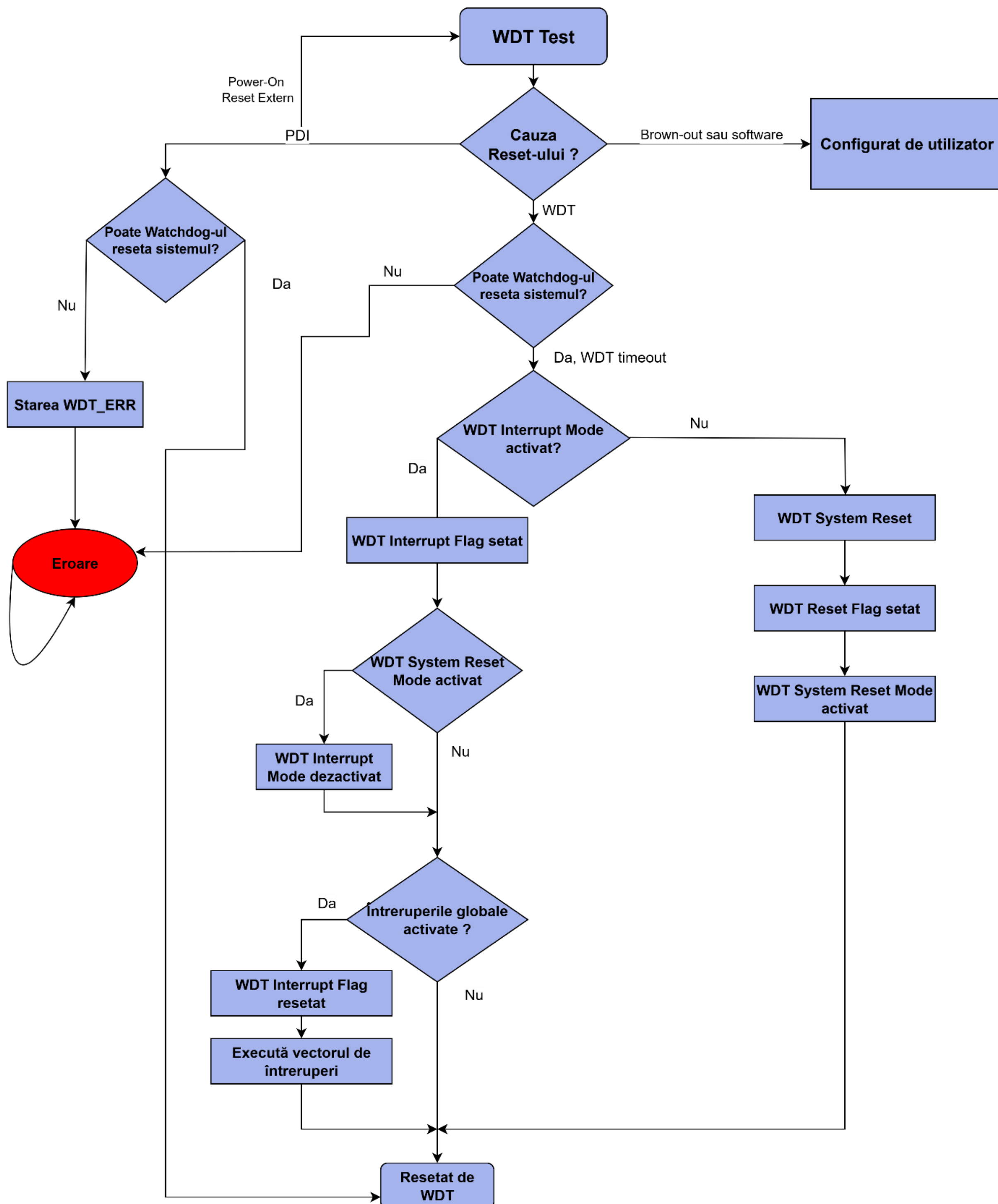


Figura 7.3 – Diagrama de test

Fluxul de execuție fără erori:

1. După punerea sub tensiune sau reset extern, se verifică dacă WDT-ul poate reseta sistemul.
2. Se resetează starea **WDT_1**, sistemul fiind resetat de WDT.
3. Se verifică dacă WDT-ul poate fi resetat, se resetează starea **WDT_2** și sistemul este resetat de WDT.
4. Se verifică dacă modul window funcționează corect, se setează **WDT_3** și sistemul este resetat de WDT.
5. Se configurează WDT-ul, se setează starea testului **WDT_OK** și se continuă cu funcția main.

Primul pas este asigurarea faptului că WDT-ul poate genera **semnalul de reset**. Acest lucru se realizează configurând Watchdog-ul conform **datasheet-ului** și așteptând până când procesul de resetare a avut loc. În plus RTC-ul este folosit pentru a estima perioada watchdog-ului, care este necesară în fazele ulterioare ale testului.

Acest lucru se realizează setând perioada RTC-ului cu o valoare destul de mică (**aproximativ 3 ms**) și numărând perioadele RTC-ului până la **resetare**. Există un timp maxim de așteptare care poate fi configurat, după expirarea acestui timp programul intră într-o **stare de eroare**.

Al doilea pas este asigurarea faptului că WDT-ul poate fi **resetat** și verificarea perioadei watchdog-ului. Starea de eroare este setată temporar, iar apoi se verifică dacă perioada WDT-ului este **mai mare** decât un minim stabilit. Se verifică dacă **diferența de frecvență** între WDT și RTC este situată într-un interval care ne asigură că ambele module funcționează după așteptări. Apoi WDT-ul este configurat, se folosește RTC pentru a aștepta ¾ din perioada watchdog-ului, care a fost estimată la pasul anterior, astfel se verifică dacă perioada WDT-ului nu expiră mai devreme decât se așteaptă. După aceea WDT-ul este **resetat**, iar programul așteaptă din nou ¾ din perioada WDT-ului. Dacă mecanismul de resetare a watchdog-ului a decurs fără probleme, sistemul ar trebui să reseteze în timp ce programul este în a doua așteptare. Acest lucru se datorează faptului că perioada totală de așteptare este **1.5** din perioada totală a WDT-ului estimată anterior. În plus, această resetare prematură ar duce testul în starea de eroare. Presupunând că WDT-ul a fost resetat corect, sistemul trebuie să fie **resetat** în aproximativ ¼ din perioada WDT-ului. Testul trece în starea următoare, iar programul este în așteptarea resetului.

Al treilea pas este asigurarea faptului că WDT-ul **funcționează corect** în modul **fereastră**. Acest lucru are la bază setarea WDT-ului și trecerea testului în următoarea stare iar apoi resetarea lui prematură. Având în vedere că perioada nu este respectată WDT-ul ar trebui să genereze un semnal de reset. Astfel, după resetarea prematură a WDT-ului programul așteaptă resetarea sistemului ¼ din perioada totală a watchdog-ului. Dacă a apărut o problemă, sistemul **nu se va reseta**, iar programul va semnala **starea de eroare**.

Al patrulea și ultimul pas: programul pur și simplu setează WDT-ul în modul **fereastră**, iar testul în starea **WDT_OK**. După aceasta, aplicația este responsabilă de resetarea WDT-ului în funcție de stări. Dacă testul se află în starea de eroare, va fi apelat un sistem de tratare a erorilor predefinit de utilizator. În mod implicit, dispozitivul va fi **”suspendat”** deoarece un WDT funcțional este **esențial** pentru o aplicație software sigură. RTC-ul și variabilele de stare sunt declarate în așa fel încât compilatorul nu le inițializează după resetare. Acest lucru permite folosirea lor pe durata tuturor resetărilor.

7.8.3 TESTAREA RUTINEI DE AUTODIAGNOSTICARE

În primul rând, rutina de autodiagnosticare va seta **starea de eroare** dacă nu are loc **resetarea sistemului**. O posibilă problemă care poate împiedica WDT-ul să genereze un semnal de reset ar putea fi **dezactivarea** lui. Starea de eroare va fi stabilită, iar sistemul pur și simplu va fi ”suspendat”.

În cazul în care Watchdog-ul **nu are** defecte de producție și resetarea sistemului este generată de acesta, ar trebui identificat cum se produce aceasta, și anume, dacă este precedată de o întrerupere sau nu. Resetarea are loc în **modul de reset**, dar și în **al treilea mod de operare** al Watchdog-ului care le combină pe primele două. Prima dată se verifică dacă modul de întreruperi este activat, iar în caz contrar se setează **flag-ul de reset** și se activează modul corespunzător. În caz afirmativ, **flag-ul de întreruperi** este activat, iar apoi se verifică și dacă modul de reset este **activat**, caz în care se **dezactivează** modul de întreruperi, apoi verificându-se întreruperile globale. În caz afirmativ, se execută **vectorul de întreruperi** și se resetează **flag-ul asociat**.

7.9 DESCRIEREA REGIŞTRILOR

7.9.1 REGISTRUL MCUSR - MCU STATUS REGISTER

Regisrul de stare **MCUSR** furnizează informații cu privire la sursa de reset care a resetat microcontrolerul.

Bit	7	6	5	4	3	2	1	0	
(0x60)	-	-	-	JTRF	WDRF	BORF	EXTRF	PORF	MCUSR
Read/Write	R	R	R	R/W	R/W	R/W	R/W	R/W	
Initial value	0	0	0	See Bit Description					

Figura 7.4 – Registrul de stare MCUSR

- **Bit 0 – PORF: Power-on Reset Flag**
  Acest bit este setat (este scrisă valoarea 1 logic) în cazul unui Power-on Reset și resetat scriind valoarea 0 în dreptul său.

- **Bit 1 – EXTRF: External Reset Flag**
  Acest bit este setat în cazul unui reset extern și resetat (este scrisă valoarea 0 logic) în cazul unui Power-on Reset sau dacă este scrisă valoarea 0 în dreptul său.

- **Bit 2 – BORF: Brown-out Reset Flag**
  Fenomenul de Brown-out constă în scăderea valorii tensiunii sub un anumit prag într-un circuit electronic. Acest bit este setat în cazul unui Brown-out Reset și resetat în cazul unui Power-on Reset dacă este scrisă valoarea 0 în dreptul său.

- **Bit 3 – WDRF: Watchdog Reset Flag**
  Acest bit este setat în cazul unui Watchdog Reset și resetat în cazul unui Power-on Reset dacă este scrisă valoarea 0 în dreptul său.

- **Bit 4 – JTRF: JTAG Reset Flag**
  Acest bit este setat dacă resetarea a fost cauzată de registrul de reset al JTAG-ului selectat de instrucțiunea AVR_RESET, fiind necesar ca registrul să aibă valoarea 1 logic. Este resetat în cazul unui Power-on Reset sau dacă este scrisă valoarea 0 în dreptul său.

Pentru identificarea corectă a tipului de reset, utilizatorul ar trebui să citească și să reseteze registrul MCUSR cât mai devreme posibil în program. În contextul în care se întâmplă acest lucru, se poate identifica tipul reset-ului examinând flag-urile de reset.

7.9.2 REGISTRUL WDTCSR - WATCHDOG TIMER CONTROL REGISTER

Bit	7	6	5	4	3	2	1	0	
(0x60)	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0	WDTCSR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial value	0	0	0	0	X	0	0	0	

Figura 7.5 – Registrul WDTCSR

- **Bit 5, 2:0 – WDP 3:0: Watchdog Timer Prescaler**
  Biții WDP 3:0 determină prescalarea Watchdog-ului când acesta este activat, adică setează numărul de cicli ai acestuia (Să se consulte **tabelul 1.2**).

- **Bit 3 – WDE : Watchdog System Reset Enable**
  Bitul WDE este suprascris de bitul WDRF în registrul MCUSR. Prin urmare, WDE este setat întotdeauna când WDRF este setat. Pentru a reseta WDE, mai întâi trebuie resetat WDRF. Acest lucru asigură resetări multiple în condițiile blocării sau nefuncționării unui program și o pornire sigură după acest lucru.
- **Bit 4 – WDCE : Watchdog Change Enable**
  Acest bit este utilizat în secvențe cronometrate pentru a schimba WDE și biții de prescalare. Pentru a reseta bitul WDE și/sau schimba biții de prescalare, WDCE trebuie setat. După ce a fost setat, hardware-ul va reseta acest bit dupa 4 cicli de ceas.

- **Bit 6 – WDIE: Watchdog Interrupt Enable**
  Când acest bit are valoarea 1 și bitul de întreruperi (I-bit) din registrul de stare (nu MCUSR) este setat, se activează întreruperea Watchdog-ului. Dacă WDE este resetat, iar I-bit-ul este activat, watchdog-ul se află în modul de întreruperi și întreruperea corespunzătoare se execută în contextul în care perioada de timp prestabilită a expirat. Dacă bitul WDE este setat Watchdog-ul se află în modul Interrupt and System Reset.

- **Bit 7 – WDIF: Watchdog Interrupt Flag**
  Acest bit este setat când perioada de timp a Watchdog-ului expiră, iar acesta este configurat pentru întreruperi. WDIF este resetat de hardware când se execută vectorul de întreruperi sau când este scrisă valoarea 1 în dreptul său.

WDTON	WDE	WDIE	Mode	Action on Time-out
1	0	0	Stopped	None
1	0	1	Interrupt Mode	Interrupt
1	1	0	System Reset Mode	Reset
1	1	1	Interrupt and System Reset Mode	Interrupt, then System Reset Mode
1	x	X	System Reset Mode	Reset

Tabelul 7.1 – Modurile de operare ale biților registrului WDTCSR

WDP	WDP2	WDP1	WDP0	Number of WDT Oscillator Cycles	Typical Time-out at $V_{CC} = 5.0$ V
0	0	0	0	2K (2048 cycles)	16ms
0	0	0	1	4K (4096) cycles	32ms
0	0	1	0	8K (8192) cycles	64ms
0	0	1	1	16K (16384) cycles	0.125s
0	1	0	0	32K (32768) cycles	0.25s
0	1	0	1	64K (65536) cycles	0.5s
0	1	1	0	128K (131072) cycles	1.0s
0	1	1	1	256K (262144) cycles	2.0 s
1	0	0	0	512K(524288) cycles	4.0s
1	0	0	1	1024K (1048576) cycles	-
1	0	1	0	-	-
1	0	1	1	-	-
1	1	0	0	-	-
1	1	0	1	-	-
1	1	1	0	-	-
1	1	1	1	-	-

Tabelul 7.2 – Tabela de adevăr a biților WDP

7.10 PROBLEME

Pentru aplicațiile **1.10.3**, **1.10.4** și **1.10.5**, se pun la dispoziție bibliotecile: **`usart.h`**, **`usart.c`**, **`round_buff.h`**, **`round_buff.c`**, **`myprint.h`** și **`myprint.c`** din **capitolul USART.**

7.10.1 RESETAREA WATCHDOG-ULUI

Cerință: Să se creeze o aplicație demonstrativă pentru **Watchdog Timer** (**WDT**) care să ilustreze resetarea automată a microprocesorului în cazul unui **blocaj software**. Aplicația va configura și activa **WDT-ul** cu un timp de expirare prestabilit. Ulterior, microprocesorul va fi **"blocat"** într-o stare de așteptare infinită, permițând **Watchdog Timer-ului** să expire și să inițieze o resetare a sistemului.

Sugestii: *În primul rând se va activa* ***Watchdog Timer-ul (atenţie şi la timpul de expirare)****. În al doilea rând se va* ***"bloca"*** *microprocesorul, de exemplu într-o* ***buclă infinită****, lăsând Watchdog Timer-ul să expire. Pentru a observa din exterior pornirea din nou a aplicaţiei (ceea ce se va întâmpla după resetarea microprocesorului), se poate, de exemplu, să se seteze şi apoi să se reseteze un pin de ieşire, de exemplu* ***PE6****, înainte de a "bloca" microprocesorul. Deoarece pinul* ***PE6*** *va fi urmărit pe o durată de timp de ordinul sutelor de milisecunde, este posibil durata cât acesta va avea valoarea* ***1 logic*** *să fie inobservabilă pe osciloscop. Din acest motiv este binevenită introducerea unei* ***întârzieri*** *înainte de a reseta pinul* ***PE6****. De asemenea, se recomandă introducerea unui* ***breakpoint*** *în dreptul liniei de cod* ***`MCUSR = 0`*** *pentru a vizualiza biții regisrului la resetarea realizată de* ***Watchdog****.*

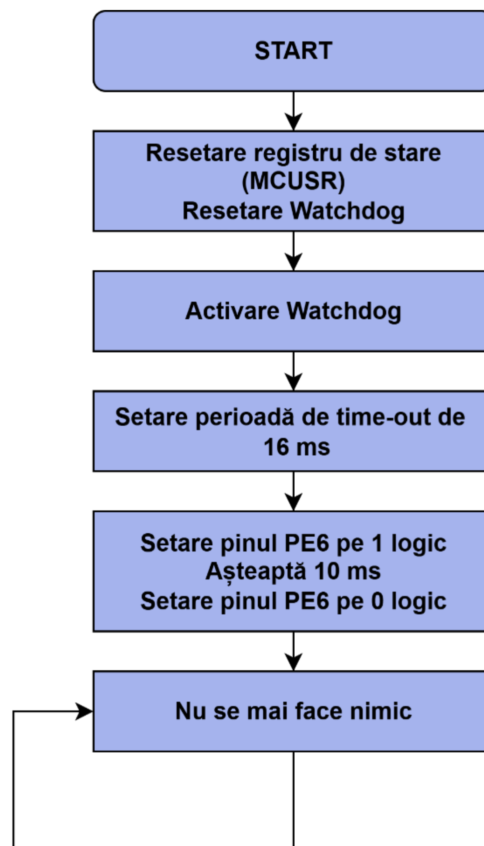


Figura 7.6 – Schema logică a aplicației 1.10.1

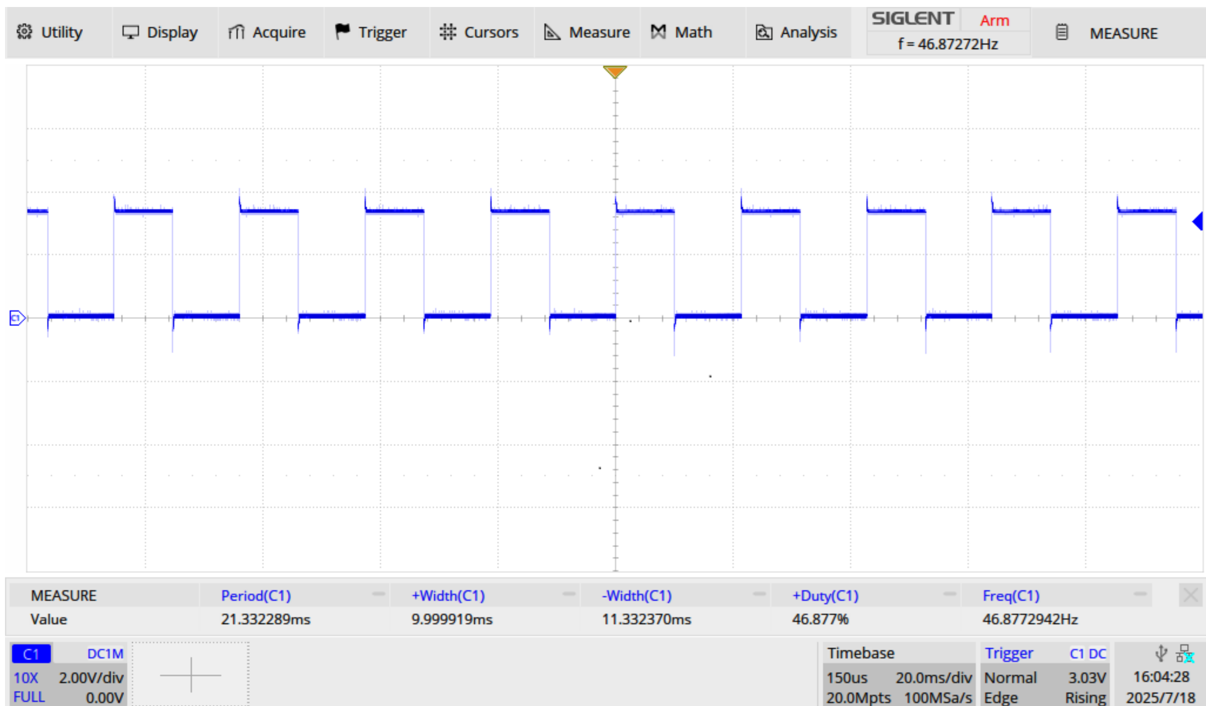


Figura 7.7 – Rezultatele obținute pe osciloscop la rezolvarea aplicației 1.10.1

`main.c`

`main.c` este punctul de intrare al aplicației, unde se inițializează și activează **Watchdog Timer-ul** (**WDT**) cu o perioadă de timeout implicită. Un pin de ieșire (**PE6**) este configurat și setat temporar la **1** logic, apoi la **0** logic, servind ca indicator vizual al repornirii sistemului. Aplicația intră apoi într-o buclă infinită (**`while(1)`**), permițând **WDT-ului** să expire și să reseteze microprocesorul, demonstrând astfel funcționalitatea sa de siguranță.

```
/*----------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul principial de rulare a aplicației Reset & Watchdog
 *--------------------------------------------------------------------*/

/*----------------------------------------------------------------------
 * Includes
 *--------------------------------------------------------------------*/

// General
#include <iom1280.h>
#include <inavr.h>

/*----------------------------------------------------------------------
 * Public defines
 *--------------------------------------------------------------------*/

// Delay Cycles
#define DELAY_CYCLES 160000

void main(void)
{
  // Resetarea registrului de stare
  MCUSR = 0;
```

```
  // Resetarea Watchdog-ului
  WDTCSR |= (1 << WDCE) | (1 << WDE);
  WDTCSR = 0;

  // Activarea Timer-ului Watchdog
  WDTCSR |= (1 << WDCE) | (1 << WDE);

  // Setarea perioadei implicite de time-out de 16 ms
  WDTCSR = (1 << WDE);

 // Setarea PE6 ca pin de ieșire
  DDRE |= (1 << PE6);

 // Setarea PE6 pe 1 logic
  PORTE |= (1 << PE6);

 __delay_cycles(DELAY_CYCLES);

 // Setarea PE6 pe 0 logic
  PORTE &= ~(1 << PE6);

 while(1);
}
```

7.10.2 DETERMINAREA PERIOADEI DE TIMEOUT

Cerință: Să se creeze o care să determine **perioada efectivă** de time-out a **Watchdog Timer-ului** (**WDT**). Aplicația va dezactiva orice **WDT** activ, va configura un nou **WDT** cu un timp de expirare prestabilit (ex. 32 ms) și va folosi pinul **PB7** pentru a genera un semnal măsurabil cu osciloscopul. Înainte de resetarea provocată de **WDT**, microcontrolerul va fi blocat pentru un **număr fix de cicluri** (ex. 1.000), simulând un **blocaj software**. Perioada se va determina ca intervalul dintre primul front pozitiv și ultimul front negativ al semnalului de pe **PB7**.

Sugestii: *Se recomandă introducerea unui **delay scurt** între schimbările de stare ale pinului **PB7** pentru a fi mai ușor vizibil pe osciloscop. Configurarea **WDT-ului** se va face prin setarea corespunzătoare a biților **WDCE**, **WDE** și **WDPx** din registrul **WDTCSR**. Pentru verificarea tipului de resetare, se poate introduce un **breakpoint** la linia `MCUSR = 0` și se pot examina valorile biților corespunzători din registrul **MCUSR**. În cazul unei perioade foarte mici a **WDT-ului** (ex. 32 ms), semnalul de pe **PB7** va oscila rapid; dacă se dorește o perioadă mai mare, trebuie modificată configurația bitilor **WDPx**.*

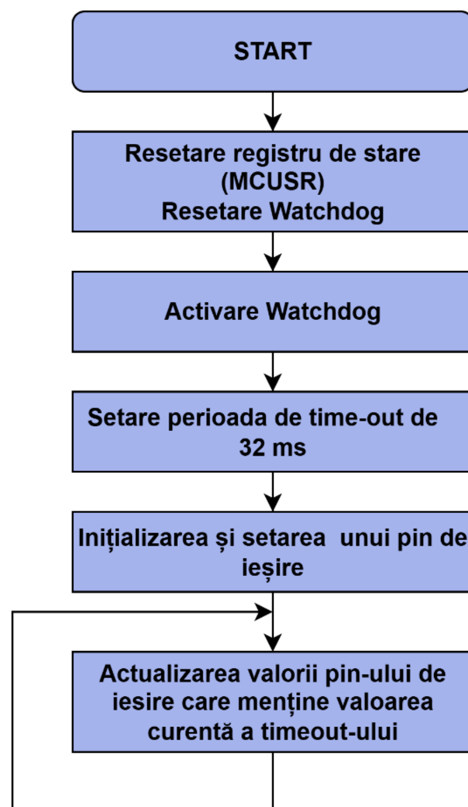


Figura 7.8 – Schema logică a aplicației 1.10.2

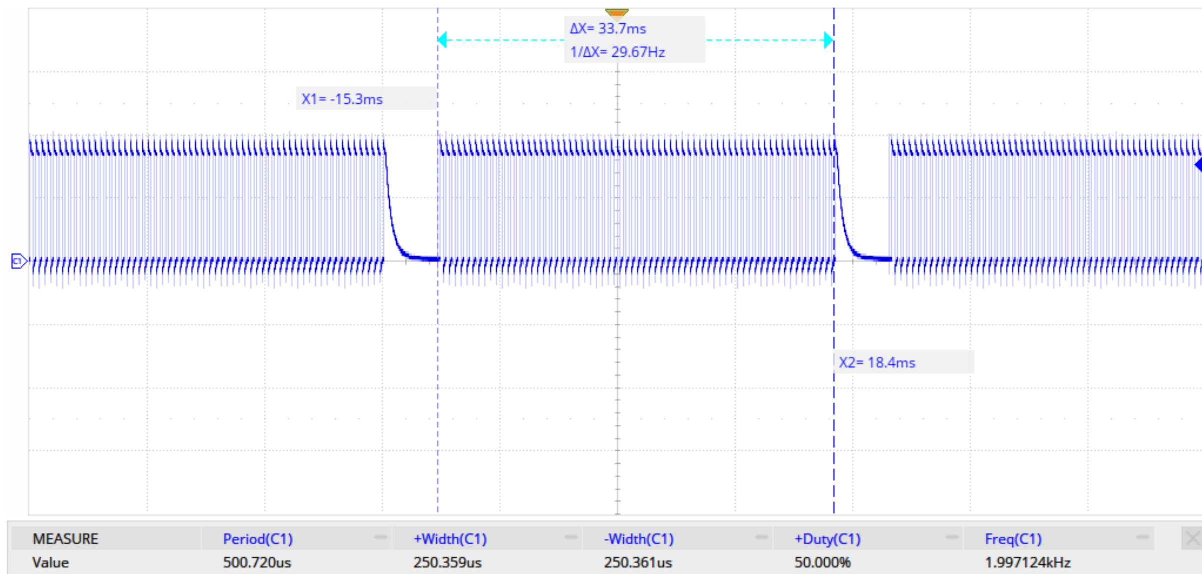


Figura 7.9 – Rezultatele obținute pe osciloscop la rezolvarea aplicației 1.10.2

`main.c`

`main.c` este punctul de intrare al aplicației, unde se dezactivează orice **Watchdog Timer** (**WDT**) activ anterior și se resetează registrul de stare **MCUSR** pentru a șterge cauza precedentă a resetării. Ulterior, **WDT-ul** este inițializat și activat cu o perioadă de timeout de **32 ms** prin configurarea biților corespunzători din registrul **WDTCSR**. Pinul de ieșire **PB7** este configurat ca ieșire digitală și setat inițial la nivelul logic **1**. În bucla infinită **`while(1)`**, acest pin își schimbă periodic starea (**toggle**), generând un semnal dreptunghiular care poate fi măsurat cu ajutorul unui osciloscop, permițând determinarea aproximativă a perioadei efective a **WDT-ului**.

```
/*-----------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul principal de rulare a aplicației Reset & Watchdog
 *---------------------------------------------------------------------*/

/*-----------------------------------------------------------------------
 * Includes
 *---------------------------------------------------------------------*/
#include <iom1280.h>
#include <inavr.h>

int main( void )
{
 __delay_cycles(1000);
 unsigned char c = WDTCSR;

 // Resetarea registrului de stare
 MCUSR = 0;

 // Restarea bitului WDE din registrul WDTCSR
 c &= ~(1 << WDE);

 // Activarea bitului WDCE din WDTCSR
 WDTCSR |= (1 << WDCE);
 WDTCSR = c;

 // Resetarea WDT
 asm("WDR");
```

```
 // Inițializarea Watchdog și setarea perioadei de time-out de 32 ms
 WDTCSR |= (1 << WDCE) | (1 << WDE);
 WDTCSR = (1 << WDE) | (1 << WDP0);

 // Inițializarea și setarea bitului de ieșire
 DDRB |= (1 << PB7);
 PORTB |= (1 << PB7);

 while(1)
 {
  // Schimbarea valorii bitului de ieșire
   PORTB ^= (1 << PB7);
   __delay_cycles(4000);
 }
 return 0;
}
```

7.10.3 CALCULAREA FRECVENȚEI TIMER-ULUI

Cerință: Să se creeze o aplicație care să calculeze frecvența efectivă a **Watchdog Timer-ului** (**WDT**) folosind unul dintre timerele interne ale microcontrolerului. La activarea **WDT-ului**, se va porni un timer **hardware**, iar după resetarea automată generată de acesta, se vor folosi valorile stocate ale timer-ului pentru a determina perioada reală de time-out. Frecvența va fi calculată pe baza acestei perioade și transmisă prin **interfața serială**, fără a utiliza tipul de date `double`.

Sugestii: *Se vor folosi variabile declarate cu atributul* `__no_init` *pentru a păstra valorile timer-ului și numărul de overflow-uri chiar și după resetarea microcontrolerului.* ***Timer-ul*** *va fi configurat în mod* ***normal****, cu un prescaler potrivit și întrerupere la overflow pentru a număra depășirile. La prima execuție, aplicația va porni* ***WDT-ul*** *și* ***timer-ul****. După resetarea cauzată de* ***WDT****, valorile salvate vor fi folosite pentru a calcula perioada efectivă a* ***WDT-ului*** *conform relației* ***T = 1 / f****. Calculul se va face în unități convenabile (milisecunde, nanosecunde, kHz), iar rezultatul va fi transmis pe serială folosind funcții dedicate. După transmiterea rezultatului, timer-ul și variabilele vor fi resetate, iar ciclul de măsurare se poate repeta.*

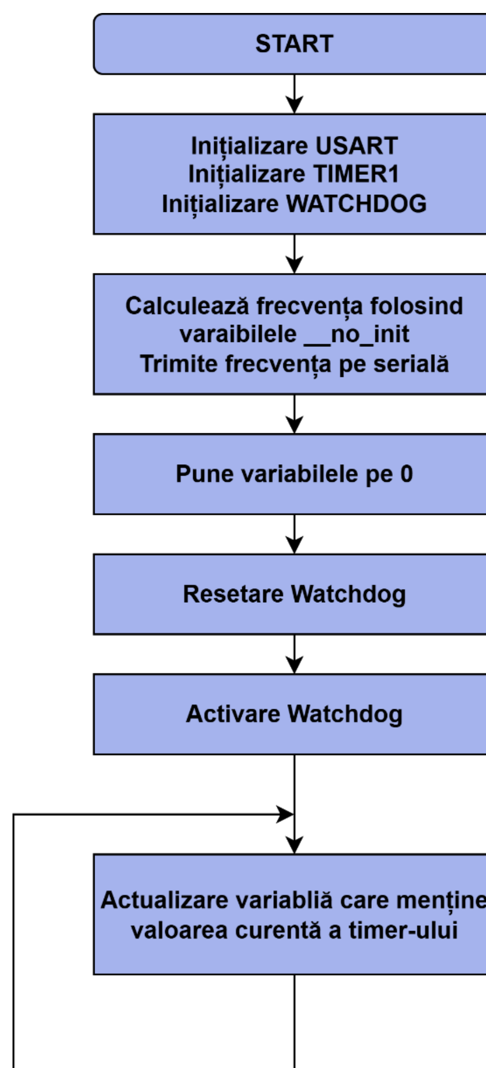


Figura 7.10 – Schema logică a aplicației 1.10.3

main.c

main.c este punctul de intrare al aplicației, unde se configurează **Timer1** pentru a măsura perioada efectivă a **Watchdog Timer-ului** (**WDT**) setat cu un timeout de **32 ms**. Variabilele **__no_init** păstrează valorile după reset, permițând calculul frecvenței **WDT** la repornire. Rezultatul este transmis prin **interfața serială**, apoi contoarele și timerul sunt resetate, iar aplicația reinițializează **WDT-ul** și actualizează continuu valorile **Timer1** într-o buclă infinită.

```
/*---------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul principal de rulare a aplicației de calcul al frecvenței
 *-------------------------------------------------------------------*/

/*---------------------------------------------------------------------
 * Includes
 *-------------------------------------------------------------------*/
#include <iom1280.h>
#include <inavr.h>
#include <stdint.h>
#include "mylib.h"

/*---------------------------------------------------------------------
 * Variabile globale
 *-------------------------------------------------------------------*/

/*
 * Variabila pentru valoarea curentă a Timer1, reținută după resetare
 * (fără inițializare)
 */
__no_init uint16_t Timer1_currentValue;

/*
 * Variabila pentru numărul de overflow-uri ale Timer1, reținută după
 * resetare
 */
__no_init uint8_t Timer1_numberOverflows;

// Rutina de întrerupere pentru overflow-ul Timer1
#pragma vector = TIMER1_OVF_vect
__interrupt void T1_OVF()
{
    // Incrementarea numărului de overflow-uri
    Timer1_numberOverflows++;
}

int main(void)
{
    // Resetarea registrului MCUSR (șterge flag-urile de reset)
    MCUSR = 0;

    // Inițializarea USART-ului pentru transmiterea datelor serial
    USART_initialize(BAUD_RATE);

    /*-----------------------------------------------------
     * Inițializarea Timer1 în mod Normal
     * Prescaler setat pe CS10 (fără divizarea frecvenței)
     *---------------------------------------------------*/
    TCCR1B |= (1 << CS10);    // Start Timer1 cu prescaler 1
```

```
    // Activarea întreruperii de overflow pentru Timer1
    TIMSK1 |= (1 << TOIE1);

    // Activarea întreruperilor globale
    __enable_interrupt();

    /*-----------------------------------------------------
     * Inițializarea și activarea Watchdog Timer
     * Setare perioadă de timeout: 32 ms
     *-----------------------------------------------------*/
    WDTCSR |= (1 << WDCE) | (1 << WDE);    // Permite modificarea WDT
   WDTCSR = (1 << WDE) | (1 << WDP0);     // Activare WDT cu WDP0 (32 ms)

    /*-----------------------------------------------------
     * Dacă există valori salvate în variabile după resetare, calculăm
     * perioada efectiva a WDT
     *-----------------------------------------------------*/
    if (Timer1_numberOverflows > 0 || Timer1_currentValue > 0)
    {
        // Calcularea numărului total de ticks
        uint32_t number = (Timer1_numberOverflows * (uint64_t)65535) +
        Timer1_currentValue;

        // Conversia în milisecunde (fiecare tick durează 62.5 ns)
        uint8_t period = number * 0.0000625;

        // Calcularea duratei unui ciclu de clock al WDT (în nanosecunde)
        uint16_t time_per_clock = (period * 1000000) / 4096;

        // Calcularea frecvenței WDT în kHz
        uint8_t frecv = (1000000. / time_per_clock);

        // Copierea frecvenței într-o variabilă pentru transmitere
        uint16_t copyFrecv = frecv;
        myprint(INTEGER, &copyFrecv);  // Trimiterea valorii prin serială
    }

    /*-----------------------------------------------------
     * Resetarea valorilor pentru următoarea măsurătoare
     *-----------------------------------------------------*/
    Timer1_numberOverflows = 0;
    Timer1_currentValue = 0;
    TCNT1 = 0;

    // Resetarea Watchdog-ului
    asm("WDR");

    // (Opțional) Reactivare WDT
    // WDTCSR |= (1 << WDE);

    /*-----------------------------------------------------
     * Bucla principală
     * Salvează permanent valoarea curentă a Timer1
     *-----------------------------------------------------*/
    while (1)
    {
        Timer1_currentValue = TCNT1;
    }

    return 0;
}
```

7.10.4 MODUL INTERRUPT AND SYSTEM RESET

Cerință: Să se realizeze o aplicație demonstrativă care să evidențieze **modul al treilea de funcționare** al **Watchdog Timer-ului** (**WDT**), și anume modul „**Interrupt and System Reset**”. Aplicația va configura **WDT-ul** cu o perioadă de timeout de **4 secunde** și va utiliza interfața serială pentru comunicare. În timpul funcționării, dacă utilizatorul trimite caractere prin serială, **WDT-ul** va fi resetat, prevenind resetarea sistemului. Dacă nu se primește niciun caracter pe serială pentru o durată **mai mare** decât timeout-ul Watchdog-ului, acesta va declanșa mai întâi o **întrerupere** (în care se va transmite caracterul **'!'**), iar apoi va reseta sistemul. Aplicația va transmite inițial mesajul **„Salut”** pe serială pentru a semnaliza pornirea programului.

Sugestii: *Se va configura și activa* ***Watchdog Timer-ul*** *în modul „****Interrupt and System Reset****” cu timeout de 4 secunde. Se vor activa întreruperile globale și cele pentru Watchdog. În rutina de întrerupere Watchdog se va trimite caracterul* ***'!'*** *pe serială pentru a semnala expirarea timeout-ului înainte de resetare. În bucla principală, se va monitoriza serialul pentru caractere primite; la fiecare caracter recepționat, Watchdog-ul se va reseta.*

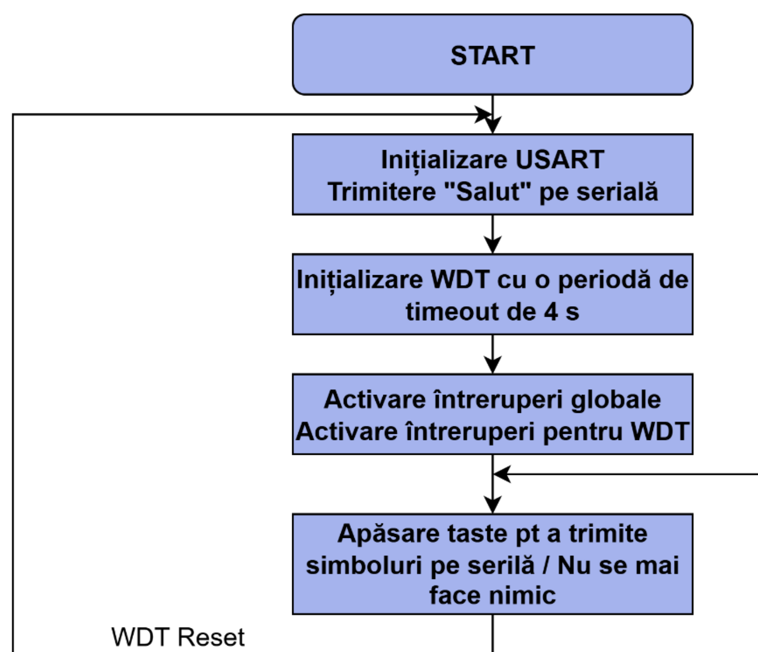


Figura 7.11 – Schema logică a aplicației 1.10.4

`main.c`

Fișierul **`main.c`** configurează și activează **Watchdog Timer-ul** în modul **„Interrupt and System Reset”** cu timeout de 4 secunde. Prin intermediul **USART-ului**, trimite mesajul inițial **„Salut”** și retransmite caracterele primite de la utilizator, resetând **Watchdog-ul** la fiecare caracter recepționat. Dacă nu se primesc date, **Watchdog-ul** declanșează întâi o întrerupere în care se transmite caracterul **‘!’**, iar apoi resetează sistemul, demonstrând funcționarea modulului de siguranță.

```
/*----------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul principial de rulare a aplicației
 *--------------------------------------------------------------------*/

/*----------------------------------------------------------------------
 * Includes
 *--------------------------------------------------------------------*/

#include <iom1280.h>
#include <inavr.h>
#include <stdint.h>
#include "mylib.h"
#include "usart.h"

/*----------------------------------------------------------------------
 * Vector de întrerupere Watchdog Timer
 *--------------------------------------------------------------------*/
#pragma vector = WDT_vect
__interrupt void intrerupere()
{
  uint8_t c = '!';
/*
 * La declanșarea întreruperii Watchdog se trimite caracterul '!'
 * prin USART
 */
  myprint(CHAR, &c);
}

int main(void)
{
  // Inițializarea USART-ului cu baudrate-ul definit în "usart.h"
  USART_initialize(BAUD_RATE);

  // Mesaj inițial de transmis
  uint8_t message[20];
  message[0]  = 'S';
  message[1]  = 'a';
  message[2]  = 'l';
  message[3]  = 'u';
  message[4]  = 't';
  message[5]  = '\0';

  // Transmiterea șirului de caractere prin USART
  USART_transmit_string(message, 6);

/*
 * Resetarea registrului de stare pentru a curăța flag-urile de
 * resetare
 */
  MCUSR = 0;
```

```
  // Resetarea Watchdog-ului înainte de configurare
  // Setarea bit-ului pentru modificarea WDT
  WDTCSR |= (1 << WDCE) | (1 << WDE);
  // Dezactivare Watchdog
  WDTCSR = 0;

  // Activarea Watchdog Timer-ului
  WDTCSR |= (1 << WDCE) | (1 << WDE);

  // Setarea perioadei de time-out de aproximativ 4 secunde
  // WDP3 activează intervalul mai lung
  WDTCSR = (1 << WDE) | (1 << WDP3);

  // Activarea întreruperii globale
  __enable_interrupt();

  // Activarea întreruperii Watchdog (Watchdog Interrupt Enable)
  WDTCSR = (1 << WDIE);

  // Bucla infinita principală
  while(1)
  {
    uint8_t aux;

    // Primirea caracter USART (funcție blocantă)
    USART_receive_char(&aux);

    if(aux != '\0')
    {
  // Resetarea Watchdog-ului pentru a evita resetarea microcontrolerului
      __watchdog_reset();

      /* Reactivarea întreruperii Watchdog
      WDTCSR = (1 << WDIE);

      /* Retrimiterea caracterului primit prin USART
      USART_transmit_char(aux);
    }
  }

  return 0;
}
```

7.10.5 IDENTIFICAREA SURSEI RESET-ULUI

Cerință: Să se realizeze o aplicație demonstrativă care să identifice sursa ultimei resetări a microcontrolerului utilizând registrul **MCUSR**. În funcție de cauza resetării (**Power-on Reset**, **External Reset**, **Brown-Out Reset**, **Watchdog Reset** sau **JTAG Reset**), aplicația va aprinde unul sau mai multe **LED-uri** de pe placă pentru a semnaliza tipul evenimentului. În cazul unui **Power-on Reset** vor fi aprinse **toate** cele 4 LED-uri, iar pentru celelalte tipuri de resetare se va aprinde doar LED-ul asociat.

Sugestii: *Se va analiza registrul **MCUSR** și biții săi pentru determinarea tipului de resetare. Se vor configura pinii aferenți LED-urilor ca ieșiri și se vor seta în funcție de bitul activ din **MCUSR**. După detectarea tipului de resetare, **MCUSR** va fi **resetat** pentru a șterge flag-urile și a pregăti sistemul pentru o nouă pornire. Aplicația va rămâne **blocată** în bucla principală după afișarea tipului de resetare.*

`main.c`

Fișierul **`main.c`** detectează tipul ultimei resetări a microcontrolerului, analizând registrul **MCUSR**. În funcție de cauza resetării (**Power-on**, **External**, **Brown-Out**, **Watchdog** sau **JTAG**), sunt aprinse unul sau mai multe LED-uri specifice de pe placă. După afișarea vizuală a rezultatului, aplicația **șterge** flag-urile din **MCUSR** și rămâne **blocată** într-o buclă infinită, păstrând LED-urile aprinse pentru diagnostic.

```
/*-----------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul principal al aplicației de detectare a tipului de reset
 *---------------------------------------------------------------------*/

/*-----------------------------------------------------------------------
 * Includes
 *---------------------------------------------------------------------*/

// General
#include <iom1280.h>
#include <inavr.h>
#include <stdint.h>
#include "mylib.h"
#include "usart.h"

/*-----------------------------------------------------------------------
 * Funcția principală
 *---------------------------------------------------------------------*/
int main(void)
{
  // Verificarea tipului de resetare pe baza registrului MCUSR

  // POWER-ON RESET
  if ((MCUSR & 0x01) == 0x01){
  /*
   * Configurarea LED A (PA5), LED B (PA6), LED C (PA7), LED D (PD7) ca
   * ieșiri
   */
    DDRA |= (1 << PA5) | (1 << PA6) | (1 << PA7);
    DDRD |= (1 << PD7);

    // Aprinderea LED A, LED B, LED C și LED D
    PORTA |= (1 << PA5) | (1 << PA6) | (1 << PA7);
    PORTD |= (1 << PD7);
  }
  // EXTERNAL RESET
  if ((MCUSR & 0x02) == 0x02)
  {
    // Configurarea LED B (PA6) ca ieșire
    DDRA |= (1 << PA6);
```

```
    // Aprinderea LED B
    PORTA |= (1 << PA6);
  }

  // BROWN-OUT RESET
  if ((MCUSR & 0x04) == 0x04)
  {
    // Configurarea LED C (PA7) ca ieșire
    DDRA |= (1 << PA7);

    // Aprinderea LED C
    PORTA |= (1 << PA7);
  }

  // WATCHDOG RESET
  if ((MCUSR & 0x08) == 0x08)
  {
    // Configurarea LED A (PA5) ca ieșire
    DDRA |= (1 << PA5);

    // Aprinderea LED A
    PORTA |= (1 << PA5);
  }

  // JTAG RESET
  if ((MCUSR & 0x10) == 0x10)
  {
    // Configurarea LED D (PD7) ca ieșire
    DDRD |= (1 << PD7);

    // Aprinderea LED D
    PORTD |= (1 << PD7);
  }

  // Resetarea registrului de stare MCUSR pentru ștergerea flag-urilor
  MCUSR = 0;

  // Buclă infinită pentru menținerea stărilor LED-urilor
  while (1)
  {
    // Aplicația rămâne aici după aprinderea LED-urilor
  }

  return 0;
}
```

7.11 BIBLIOGRAFIE

1. *"8-bit AVR Microcontroller ATmega 1280 Datasheet", Microchip Technology*
2. *"Schematic for UNI Clicker", Mikroe*
3. *"Schematic for ATmega1280: SiBrain", Mikroe*
4. *"Watchdog Timer", Wikipedia*

8. CODURI REDUNDANTE CICLICE – CRC

8.1 UNDE SE FOLOSEȘTE CRC?

CRC este un algoritm folosit în principal pentru detecția erorilor în transmisia sau stocarea datelor în ceea ce privește:

- **Rețele de calculatoare:** verificarea transmiterii corecte (fără erori) ale datelor printr-o rețea;
- **Dispozitive de stocare:** verificarea integrității datelor scrise sau citite pe hard disk-uri, SSD, etc;
- **Protocoale de comunicație:** verificarea comunicării între periferice prin USB, HDLC, PPP, etc;
- **Sisteme embedded:** comunicarea între microcontrollere sau între senzori și procesoare.

8.2 CUNOȘTINȚE NECESARE

Pentru a putea parcurge acest capitol, este necesar să cunoașteți următoarele subiecte din capitolele anterioare, și nu numai:

- Aritmetica binară, în special operația **XOR**.
- Reprezentarea polinoamelor în format binar.
- Noțiuni de bază despre erorile de transmisie a datelor.
- Programare în limbajul C pentru microcontrollere AVR.
- Utilizarea mediului de dezvoltare IAR Embedded Workbench.

8.3 ABSTRACT

Acest capitol detaliază metoda de detectare a erorilor folosind **CRC** (**Cyclic Redundancy Check**). Se prezintă atât fundamentele teoretice, bazate pe împărțirea polinomială în aritmetică **modulo 2**, cât și aspectele practice ale implementării. Este exemplificat modul de calcul software, atât bit cu bit, cât și folosind **tabele pre-calculate**, și demonstrează generarea automată a sumei de control folosind funcționalitățile **linker-ului** din mediul **IAR / AVR**.

8.4 HARDWARE ȘI SOFTWARE NECESAR

- ATmega 1280 SiBRAIN;
- UNI clicker;
- Atmel ICE;
- IAR Embedded Workbench 7.30.5;
- Osciloscop (opțional, pentru analiza timpilor de execuție).

8.5 INTRODUCERE ÎN CRC

Definiție

Codurile Redundante Ciclice (CRC) reprezintă o metodă eficientă de detectare a erorilor pentru blocuri de date.

Fiecărui bloc de date i se atașează o valoare de control de lungime fixă, cunoscută ca și "**checksum**" sau **cod CRC**. Această valoare este **restul** unei împărțiri polinomiale a conținutului mesajului. La recepție, calculul se repetă, iar dacă noile date de control **nu corespund** cu cele recepționate, înseamnă că a apărut o eroare.

Algoritmul se numește astfel deoarece valoarea de control este o **redundanță** (mărește mesajul fără a adăuga informație utilă) și se bazează pe proprietățile **codurilor ciclice**. Acestea sunt foarte populare datorită simplității de implementare atât în hardware, cât și în software, a eficienței în detectarea erorilor comune și a analizei matematice facile.

Principiul de bază este împărțirea polinomială în aritmetică **modulo 2** (unde adunarea și scăderea sunt echivalente cu operația logică **XOR**). Mesajul este tratat ca un **polinom**, care este **împărțit** la un **polinom prestabilit**, numit **polinom generator**. Câtul împărțirii este **ignorat**, iar restul obținut constituie **codul CRC**.

Cel mai simplu exemplu de **CRC** este **bitul de paritate**, care este un **CRC** pe **1 bit** ce folosește polinomul generator **x + 1**.

8.6 SPECIFICAȚII ȘI PARAMETRI

Implementarea practică a unui **algoritm CRC** implică definirea mai multor parametri care caracterizează în mod unic procesul de calcul.

Deşi eficiente pentru detectarea erorilor accidentale, **codurile CRC** standard **NU** sunt potrivite pentru a proteja datele împotriva alterărilor intenționate:

- **Lipsa autentificării:** Un atacator poate **modifica** mesajul și **recalcula** valoarea **CRC**, astfel încât modificarea să nu fie detectată. Aplicațiile care necesită protecție la modificări intenționate trebuie să folosească **mecanisme criptografice**, precum **semnăturile digitale**.
- **Reversibilitate:** Spre deosebire de **funcțiile hash criptografice**, funcția **CRC** este **ușor reversibilă**.
- **Liniaritate: CRC** are proprietatea liniară **crc(x) ⊕ crc(y) = crc(x ⊕ y)**. Aceasta permite manipularea unui mesaj criptat și a **CRC-ului** său asociat **fără a cunoaște** cheia de criptare.

Un **sistem CRC** este definit complet de parametrii enumerați mai jos.

8.6.1 POLINOMUL GENERATOR

Acesta este elementul central al algoritmului. Adesea, **bitul cel mai semnificativ** (care este mereu **1**) este omis din reprezentarea sa numerică (coeficientul lui x^n este mereu 1, altfel polinomul nu ar mai fi de grad n).

8.6.2 REPREZENTAREA POLINOMULUI

Un polinom poate fi reprezentat numeric în mai multe moduri. De exemplu, polinomul $\mathbf{x^4 + x + 1}$ (grad 4) poate fi scris:

- **Normal (0x3):** Corespunde la **0b0011**. Se omit termenii $\mathbf{x^4}$**,** $\mathbf{x^3}$**,** $\mathbf{x^2}$ și se reprezintă biții $\mathbf{x^1}$ **și** $\boldsymbol{x^0}$. Se folosește în împărțiri unde bitul cel mai semnificativ (**MSB**) este procesat primul.
- **Inversat (0xC):** Corespunde la **0b1100**. Este reprezentarea **în oglindă** a celei normale. Se folosește în implementări unde bitul cel mai puțin semnificativ (**LSB**) este procesat primul.
- **Inversat reciproc (Koopman) (0x9):** Corespunde la **0b1001**. Reprezintă biții polinomului, dar în loc să omiți termenul x^n, omiți termenul liber x^0.

8.6.3 VALOAREA INIȚIALĂ

Registrul CRC este inițializat cu această valoare la începutul calculului. Poate fi 0 sau orice altă valoare.

8.6.4 ORDINEA BIȚILOR

Specifică dacă biții din fiecare octet al mesajului sunt procesați de la **MSB** la **LSB** (direct) sau de la **LSB** la **MSB** (inversat / reflected).

8.6.5 XOR FINAL

Valoarea calculată a **CRC-ului** este supusă unei operații **XOR** cu această valoare înainte de a fi returnată ca rezultat final.

8.7 EXEMPLU DE CALCUL

Dacă polinomul generator este de **grad n**, atunci **CRC-ul** are **n biți**. Polinomul generator are întotdeauna **n+1 biți** (de la $\mathbf{x^n}$ până la $\mathbf{x^0}$). Pentru calculul unui **CRC** pe **n biți**, se vor poziționa în partea stângă a mesajului inițial cei **n biți** ai polinomului generator. Acest lucru lasă loc pentru restul împărțirii (**CRC-ul**), care are exact $\mathbf{x^n}$ **biți**.

Se pornește de la mesajul inițial: **11010011101100.** Se completează mesajul inițial cu **n zerouri** corespunzătoare celor **n biți** de la **CRC**. Mai jos sunt prezentate calculele pentru un **CRC** pe **3 biți**:

```
11010011101100 000  <----- completăm mesajul cu cei 3 biţi
1011  <------------------- divizorul (4 biţi) = x^3 + x + 1
------------------
01100011101100 000  <----- rezultatul
```

Dacă bitul **deasupra MSB-ului (**bitul din mesaj **corespunzător poziţiei MSB a generatorului** în segmentul curent) din divizor este **0**, nu trebuie făcute calcule.

Dacă bitul din mesajul iniţial **deasupra MSB-ului** din divizor este **1**, se face un **XOR** între mesajul iniţial şi divizor.

Apoi divizorul este mutat cu o poziţie **în dreapta** şi procesul se repetă până când divizorul ajunge în partea dreaptă a mesajului iniţial. Mai jos este prezentat calculul complet:

```
11010011101100 000  <----- mesajul este completat pe 3 biţi
1011  <------------------- divizor
01100011101100 000  <----- rezultat
 1011<-------------------- divizor
00111011101100 000
  1011
00010111101100 000
   1011
00000001101100 000
       1011
00000000110100 000
        1011
00000000011000 000
         1011
00000000001110 000
          1011
00000000000101 000
           101 1
------------------
00000000000000 100  <----- rest (3 biţi)
```

Restul obţinut reprezintă **valoarea propriu-zisă** a funcţiei **CRC**. Pentru verificarea unui mesaj primit, acesta este **divizat cu polinomul generator**. Dacă nu există erori detectabile, restul obţinut trebuie să fie **0**.

```
11010011101100 100  <----- mesajul cu valoarea de control
1011  <------------------- divizor
01100011101100 100  <----- rezultat
 1011<-------------------- divizor
00111011101100 100
......
00000000001110 100
          1011
00000000000101 100
           101 1
------------------
                 0  <----- rest (3 biţi)
```

8.8 METODE DE CALCUL CRC

8.8.1 ALGORITM BIT-CU-BIT

Aceasta este implementarea **directă** a împărțirii polinomiale. Mesajul este procesat **bit cu bit**, iar pentru fiecare bit se efectuează o operație **XOR** cu polinomul generator, dacă este cazul. Deși este simplu de înțeles, este **cea mai lentă** metodă.

8.8.2 ALGORITM BAZAT PE TABELE (PRE-CALCULAT)

Pentru mărirea vitezei, se poate **pre-calcula** rezultatul împărțirii pentru toți cei **256 de octeți** posibili și stoca într-un tabel. La execuție, se procesează mesajul **octet cu octet**, folosind tabelul pentru a actualiza valoarea **CRC**, ceea ce elimină buclele de procesare per bit și **crește semnificativ** viteza.

Multe microcontrolere conțin module **hardware** dedicate pentru **calculul CRC**, oferind cea mai mare viteză. Alternativ, uneltele **software** moderne, cum ar fi **linker-ul** din IAR Embedded Workbench, pot calcula automat un **CRC** pentru o anumită secțiune de memorie (de obicei, întreaga memorie **Flash** a programului) și pot plasa rezultatul la o adresă cunoscută. Acest lucru este util pentru a verifica integritatea firmware-ului la pornire.

8.9 POINTERI ÎN IAR

8.9.1 POINTERI ȘI TIPURI DE MEMORIE

Pointerii sunt folosiţi pentru **a referi locaţia datelor**. În general, pointerii au un tip. De exemplu, un pointer de tipul **`int*`** indică către un **întreg**. În compilator, pointerul **indică** către un anumit **tip de memorie**. Tipul memoriei este specificat utilizând un **cuvânt cheie** înainte de asterisc. De exemplu un **pointer** care **indică** către un **întreg** stocat în memoria **"far"** este declarat astfel:

```
int__far* MyPtr;
```

Trebuie menţionat faptul că locaţia variabilei pointer **`MyPtr`** nu este afectată de cuvântul cheie care precede asteriscul. În exemplul următor, variabila **`MyPtr2`** este plasată în memoria **"tiny"**. Ambele variabile, **`MyPtr`** și **`MyPtr2`**, indică către o dată de tip caracter din memoria **"far"**.

```
char __far* __tiny MyPtr2;
```

Oricând este posibil, pointerii trebuie **declaraţi fără** atribute de memorie. De exemplu, toate funcţiile din biblioteca standard sunt declarate fără specificarea explicită a tipului de memorie.

8.9.2 DIFERENȚE ÎNTRE TIPURI DE POINTERI

Un pointer trebuie să conţină **informaţia necesară** pentru a specifica **locaţia** unui anumit **tip de memorie**. Acest fapt denotă că dimensiunile pointerilor sunt **diferite** pentru **diferite** tipuri de memorie. În **IAR C/C++ Compiler For AVR** este **interzisă** conversia pointerilor de tipuri diferite fără utilizarea unui **cast explicit**.

8.9.2.1 POINTERI LA FUNCȚII

Dimensiunea unui pointer la funcţie este tot timpul **16** sau **24 de biţi**, iar aceştia pot adresa **întreaga memorie**. Reprezentarea internă a unui pointer la funcţie este **adresa** de la care începe **funcţia împărţită la 2**.

În IAR sunt disponibile următoarele tipuri de pointeri la funcţii:

Cuvânt-cheie	Interval de memorie	Dimensiune pointer	Tip index	Descriere
__nearfunc	0-0x1FFFE	2 octeţi	`signed int`	Poate fi apelat de oriunde din memoria programului, dar trebuie să refere o locaţie din primii 128 kB i acelui spaţiu de memorie.
__farfunc	0-0x7FFFFE	3 octeţi	`signed long`	Poate fi apelat oriunde.

Tabelul 8.1 - Tipurile de pointeri la funcţii disponibile în IAR

8.9.2.2 POINTERI LA DATE

Pointerii la date pot avea 3 dimensiuni: **8**, **16** sau **24 biţi**. Pointerii la date disponibili sunt:

Cuvânt-cheie	Dimensiune pointer	Spaţiul de memorie	Tipul indicelui	Intervalul de memorie
__tiny	1 octet	Data	`signed char`	0x0-0xFF
__near	2 octeţi	Data	`signed int`	0x0-0xFFFF
__far	3 octeţi	Data	`signed int`	0x0-0xFFFFFF
__huge	3 octeţi	Data	`signed long`	0x0-0xFFFFFF
__tinyflash	1 octet	Code	`signed char`	0x0-0xFF
__flash	2 octeţi	Code	`signed int`	0x0-0xFFFF
__farflash	3 octeţi	Code	`signed int`	0x0-0xFFFFFF
__hugeflash	3 octeţi	Code	`signed long`	0x0-0xFFFFFF
__eeprom	1 octer	EEPROM	`signed long`	0x0-0xFF
__eeprom	2 octeţi	EEPROM	`signed int`	0x0-0xFFFF

Tabelul 8.2 - Tipurile de pointeri la date disponibile în IAR

8.10 PROBLEME

8.10.1 SETĂRI IMPLEMENTARE CRC ÎN IAR

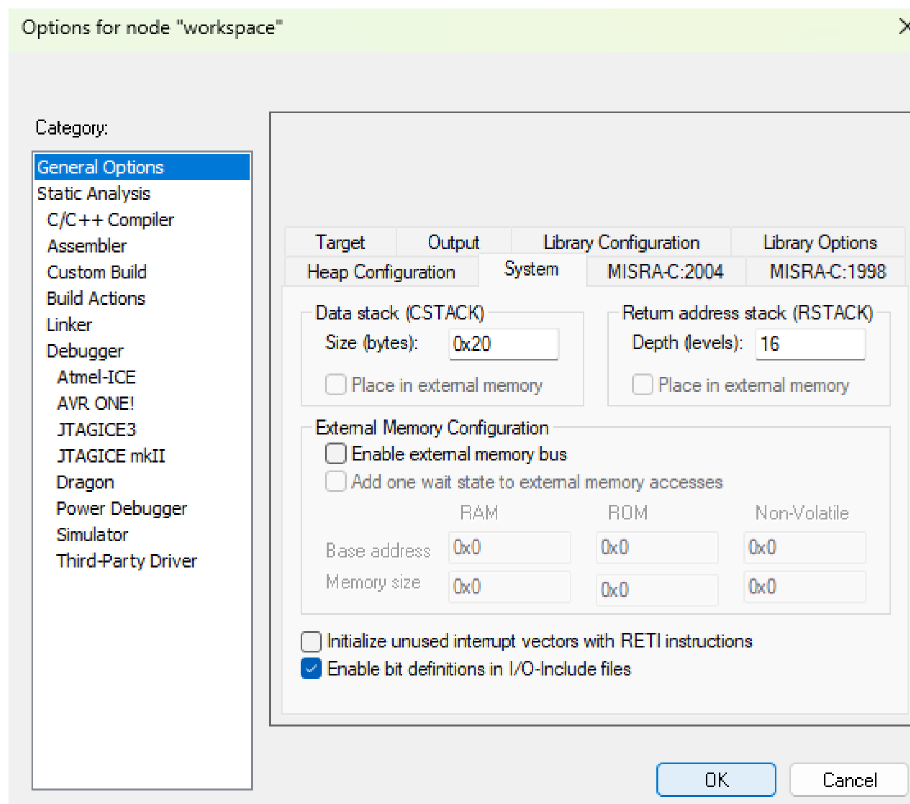


Figura 8.1 - Fereastra de opțiuni generale ale proiectului

Se lasă **debifată** căsuţa **Initialize unused interrupt vectors with RETI instructions** pentru a evita un conflict cu opţiunea **Fill unused code memory** din categoria **Linker**.

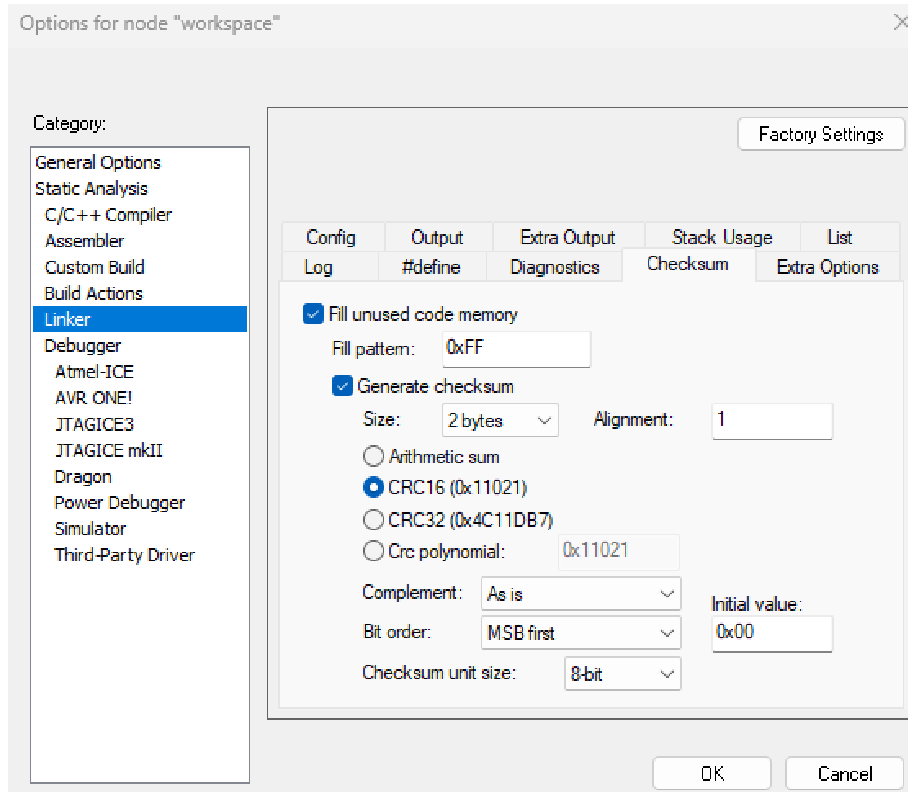


Figura 8.2 - Fereastra de opțiuni ale Linker-ului

Linker-ul din **IAR** poate fi configurat pentru a genera automat un **CRC** pentru codul programului. Acest lucru se face din **Options -> Linker -> Checksum**.

- **Fill unused code memory** – setează valoarea cu care se completează memoria neutilizată.
- **Generate checksum** – setează generarea **CRC-ului**.
- **Size** – setează dimensiunea **CRC-ului** generat în **octeți**.
- **Arithmetic sum** – calculează suma tuturor biților de **1**.
- **CRC polynomial** – setează un polinom propriu pentru generarea **CRC-ului**.
- **Complement** – setează modul de reprezentare al **CRC-ului** generat:
  - **As is** – rezultatul rămâne neschimbat.
  - **1's Complement** – complement față de **1**.
  - **2's Complement** – complement față de **2**.
- **Bit order** – modul de reprezentare al **CRC-ului** generat.
  - **LSB first** – primul bit reprezintă **coeficientul termenului** la puterea **0**.
  - **MSB first** – primul bit reprezintă **coeficientul termenului** la puterea **cea mai mare**.
  - **Initial value** – setează valoarea inițială a **CRC-ului**.

8.10.2 CRC-16

Cerință: Să se realizeze o aplicație care **calculează și verifică** valoarea **CRC-16** a unui bloc de date stocat în memoria **Flash**. Blocul de date are dimensiunea **128 KB** și este stocat în zona de adresă **0x000000 – 0x01FFFF** (spațiul Flash). Valoarea „reală" a **CRC-ului** pe 16 biți este salvată **imediat după blocul de date**, la adresele **0x020000 – 0x020003** (2 octeți CRC + un eventual padding / rezervă).
CRC-ul trebuie calculat folosind **două metode**:

1. **Metoda „bit cu bit"** (fără tabelă);
2. **Metoda cu tabele precalculate** (lookup table).

`crc16.h`

`crc16.h` reprezintă header-ul modulului de calcul al valorii **CRC pe 16 biți**, declarând funcțiile necesare pentru: calculul CRC-16 prin metoda clasică bit-cu-bit (**`crc16()`**), folosind un polinom specificat și o valoare inițială, calculul CRC-16 prin metoda cu tabelă precalculată (**`crc16wtable()`**), optimizată pentru performanță, selectarea ordinii de procesare a biților (**`enum BitOrder`** – **LSBF** sau **MSBF**), utilizarea polinoamelor standard definite pentru CRC-16 în reprezentare MSBF (**`0x1021`**) și LSBF (**`0x8408`**).

```
/*--------------------------------------------------------------------------
 * Fișier: crc16.h
 * Utilizat pentru declararea funcțiilor care calculează CRC pe 16 biți
 *------------------------------------------------------------------------*/

#ifndef _CRC_H_
#define _CRC_H_

/*--------------------------------------------------------------------------
 * Includes
 *------------------------------------------------------------------------*/

// General
#include <ioavr.h>
#include <inavr.h>
#include <stdint.h>

/*--------------------------------------------------------------------------
 * Data structures
 *------------------------------------------------------------------------*/

/*
 * Enum care definește ordinea de procesare a biților.
 * LSBF: bitul cel mai puțin semnificativ este procesat primul.
 * MSBF: bitul cel mai semnificativ este procesat primul.
 */
enum BitOrder {LSBF, MSBF};

/*--------------------------------------------------------------------------
 * Public defines
 *------------------------------------------------------------------------*/

// Polinomul standard pentru CRC-16 în reprezentare MSBF
#define CRC16_MSBF 0x1021

// Polinomul standard pentru CRC-16 în reprezentare LSBF
#define CRC16_LSBF 0x8408

/*--------------------------------------------------------------------------
 * Public (exported) functions
 *------------------------------------------------------------------------*/

/*
 * Funcția calculează CRC-16 folosind o tabelă pre-calculată (lookup table)
 * și returnează valoarea finală calculată.
 */
uint16_t crc16wtable(uint16_t init_val_16, uint32_t adr_start,
                     uint32_t len, enum BitOrder ord);
```

```
/*
 * Funcția calculează CRC-16 folosind metoda clasică bit-cu-bit (fără
 * tabelă) și returnează valoarea finală calculată.
 */
uint16_t crc16(uint16_t polinom16, uint16_t init_val_16,
               uint32_t adr_start, uint32_t len, enum BitOrder ord);

#endif
```

crc16.c

Fișierul **crc16.c** definește funcțiile și tabelele necesare pentru calculul **CRC-16** atât prin metoda clasică **bit-cu-bit**, cât și prin metoda optimizată cu **tabele precalculate** (lookup table). Conține două tabele în memorie flash, **crc16tab_MSBF** și **crc16tab_LSBF**, pentru calculul rapid al **CRC-ului** în funcție de ordinea biților (**MSB-first** sau **LSB-first**). Funcția **crc16()** implementează algoritmul de calcul **bit-cu-bit**, citind datele din memoria flash și aplicând operații de **shiftare** și **XOR** cu polinomul specificat. Funcția **crc16wtable()** utilizează tabelele precalculate pentru a accelera calculul, reducând numărul de operații la fiecare octet procesat.

```
/*----------------------------------------------------------------------------
 * Fișier: crc16.c
 * Utilizat pentru definirea funcțiilor și a tablourilor folosite de CRC-16
 *--------------------------------------------------------------------------*/

/*----------------------------------------------------------------------------
 * Includes
 *--------------------------------------------------------------------------*/

// General
#include "crc16.h"

/*----------------------------------------------------------------------------
 * Public variables
 *--------------------------------------------------------------------------*/

// Look-up table cu valori precalculate pentru calculul CRC-16 cu MSB-first
__flash const uint16_t crc16tab_MSBF[256] = {
  0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
  0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
  0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6,
  0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
  0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485,
  0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d,
  0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
  0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc,
  0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823,
  0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
  0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12,
  0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a,
  0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
  0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49,
  0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70,
  0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78,
  0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f,
  0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067,
  0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
  0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256,
  0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d,
  0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
  0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c,
  0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634,
```

```
  0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
  0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3,
  0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a,
  0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
  0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9,
  0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1,
  0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
  0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0
};

/*
 * Look-up table cu valori precalculate pentru optimizarea calculului CRC-
 * 16 cu LSB-first
 */
__flash const uint16_t crc16tab_LSBF[256] = {
0x0000, 0x1189, 0x2312, 0x329b, 0x4624, 0x57ad, 0x6536, 0x74bf,
0x8c48, 0x9dc1, 0xaf5a, 0xbed3, 0xca6c, 0xdbe5, 0xe97e, 0xf8f7,
0x1081, 0x0108, 0x3393, 0x221a, 0x56a5, 0x472c, 0x75b7, 0x643e,
0x9cc9, 0x8d40, 0xbfdb, 0xae52, 0xdaed, 0xcb64, 0xf9ff, 0xe876,
0x2102, 0x308b, 0x0210, 0x1399, 0x6726, 0x76af, 0x4434, 0x55bd,
0xad4a, 0xbcc3, 0x8e58, 0x9fd1, 0xeb6e, 0xfae7, 0xc87c, 0xd9f5,
0x3183, 0x200a, 0x1291, 0x0318, 0x77a7, 0x662e, 0x54b5, 0x453c,
0xbdcb, 0xac42, 0x9ed9, 0x8f50, 0xfbef, 0xea66, 0xd8fd, 0xc974,
0x4204, 0x538d, 0x6116, 0x709f, 0x0420, 0x15a9, 0x2732, 0x36bb,
0xce4c, 0xdfc5, 0xed5e, 0xfcd7, 0x8868, 0x99e1, 0xab7a, 0xbaf3,
0x5285, 0x430c, 0x7197, 0x601e, 0x14a1, 0x0528, 0x37b3, 0x263a,
0xdecd, 0xcf44, 0xfddf, 0xec56, 0x98e9, 0x8960, 0xbbfb, 0xaa72,
0x6306, 0x728f, 0x4014, 0x519d, 0x2522, 0x34ab, 0x0630, 0x17b9,
0xef4e, 0xfec7, 0xcc5c, 0xddd5, 0xa96a, 0xb8e3, 0x8a78, 0x9bf1,
0x7387, 0x620e, 0x5095, 0x411c, 0x35a3, 0x242a, 0x16b1, 0x0738,
0xffcf, 0xee46, 0xdcdd, 0xcd54, 0xb9eb, 0xa862, 0x9af9, 0x8b70,
0x8408, 0x9581, 0xa71a, 0xb693, 0xc22c, 0xd3a5, 0xe13e, 0xf0b7,
0x0840, 0x19c9, 0x2b52, 0x3adb, 0x4e64, 0x5fed, 0x6d76, 0x7cff,
0x9489, 0x8500, 0xb79b, 0xa612, 0xd2ad, 0xc324, 0xf1bf, 0xe036,
0x18c1, 0x0948, 0x3bd3, 0x2a5a, 0x5ee5, 0x4f6c, 0x7df7, 0x6c7e,
0xa50a, 0xb483, 0x8618, 0x9791, 0xe32e, 0xf2a7, 0xc03c, 0xd1b5,
0x2942, 0x38cb, 0x0a50, 0x1bd9, 0x6f66, 0x7eef, 0x4c74, 0x5dfd,
0xb58b, 0xa402, 0x9699, 0x8710, 0xf3af, 0xe226, 0xd0bd, 0xc134,
0x39c3, 0x284a, 0x1ad1, 0x0b58, 0x7fe7, 0x6e6e, 0x5cf5, 0x4d7c,
0xc60c, 0xd785, 0xe51e, 0xf497, 0x8028, 0x91a1, 0xa33a, 0xb2b3,
0x4a44, 0x5bcd, 0x6956, 0x78df, 0x0c60, 0x1de9, 0x2f72, 0x3efb,
0xd68d, 0xc704, 0xf59f, 0xe416, 0x90a9, 0x8120, 0xb3bb, 0xa232,
0x5ac5, 0x4b4c, 0x79d7, 0x685e, 0x1ce1, 0x0d68, 0x3ff3, 0x2e7a,
0xe70e, 0xf687, 0xc41c, 0xd595, 0xa12a, 0xb0a3, 0x8238, 0x93b1,
0x6b46, 0x7acf, 0x4854, 0x59dd, 0x2d62, 0x3ceb, 0x0e70, 0x1ff9,
0xf78f, 0xe606, 0xd49d, 0xc514, 0xb1ab, 0xa022, 0x92b9, 0x8330,
0x7bc7, 0x6a4e, 0x58d5, 0x495c, 0x3de3, 0x2c6a, 0x1ef1, 0x0f78
};

/*-----------------------------------------------------------------------
 * Public functions
 *---------------------------------------------------------------------*/

/*
 * Funcția calculează CRC-16 folosind metoda clasică bit-cu-bit (fără
 * tabelă) și returnează valoarea finală calculată.
 */
uint16_t crc16(uint16_t polinom16, uint16_t init_val_16, uint32_t
               adr_start, uint32_t len, enum BitOrder ord)
{
  uint16_t crc = init_val_16; // Inițializează CRC cu valoarea de start
```

```
  uint16_t data = 0;           // Variabilă temporară pentru octetul curent
  while(len--) {
    uint16_t i;
   // Se extrage valoarea octetului de la adresa de start din memoria flash
    data = *(__farflash char *)adr_start;
    if (ord == MSBF) // Varianta cu shiftare spre MSB
    {
      data <<= 8; // Se aliniază octetul la MSB
      crc ^= data; // Se transferă și integrează datele în CRC
      adr_start++;
      for(i = 0; i < 8; ++i) {
        if(crc & 0x8000) // Se verifică dacă MSB este 1
          crc = (crc << 1) ^ polinom16;
        else
          crc = crc << 1;
      }
    }
    else { // Varianta cu shiftare spre LSB
      crc ^= data;
      adr_start++;
      for(i = 0; i < 8; ++i) {
        if(crc & 0x0001) // Se verifică dacă LSB este 1
          crc = (crc >> 1) ^ polinom16;
        else
          crc = crc >> 1;
      }
    }
  }
  return crc;
}

/*
 * Funcția calculează CRC-16 folosind o tabelă pre-calculată (lookup table)
 * și returnează valoarea finală calculată.
 */
uint16_t crc16wtable(uint16_t init_val_16, uint32_t adr_start,
                     uint32_t len, enum BitOrder ord)
{
  uint32_t counter;
  uint32_t crc = init_val_16; // Inițializează CRC cu valoarea de start
  for( counter = 0; counter < len; counter++)
    if(ord == MSBF) // Varianta cu MSB
    /* 1. Shiftare la stânga CRC cu 8 biți
     * 2. XOR cu valoarea din tabelul MSBF corespunzătoare.
     * Indexul în tabel se obține astfel:
     * - (crc >> 8) = octetul superior al CRC curent
     * - XOR cu byte-ul citit din memorie (de la adr_start)
     * - & 0x00FF pentru a păstra doar 8 biți
     * 3. Se adună valoarea din tabel, care reprezintă efectul acelui byte
     * asupra CRC-ului.
     */
      crc = (crc << 8) ^ crc16tab_MSBF[((crc >> 8) ^ *(__farflash
            char *)adr_start++) & 0x00FF];
    else // Varianta cu LSB
      crc = (crc >> 8) ^ crc16tab_LSBF[(crc ^ *(__farflash char *)
            adr_start++) & 0x00FF];
  return crc;
}
```

main.c

Fișierul **main.c** reprezintă punctul de intrare al aplicației **CRC-16**. În cadrul acestuia, se citește valoarea reală a CRC-ului stocată în memoria flash (**real_crc**), apoi se calculează CRC-ul pentru aceeași zonă de memorie prin două metode diferite:

- **crc16()** – metoda clasică **bit-cu-bit**, care procesează fiecare bit în funcție de polinomul specificat;
- **crc16wtable()** – metoda optimizată cu **tabele precalculate** (look-up table), care reduce numărul de operații necesare pentru calculul CRC-ului.

Ambele metode procesează datele din memoria flash, cu ordinea bitilor specificată (MSB-first). Programul nu conține o buclă de execuție funcțională, rămânând blocat în bucla infinită **while(1)** după efectuarea calculelor, permițând astfel analizarea și compararea rezultatelor obținute prin cele două tehnici.

```
/*---------------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul principial de rulare a aplicației CRC-16
 *-------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------
 * Includes
 *-------------------------------------------------------------------------*/

// General
#include "crc16.h"

void main( void )
{
   /*
    * Se calculează atât CRC-ul folosind funcția standard crc16(), cât și cu
    * crc16wtable(), versiunea cu look-up table.
    */
   uint16_t real_crc=*(__farflash uint16_t *)(0x020000-2);
   uint16_t my_crc16=crc16(CRC16_MSBF,0,0,(0x020000-2),MSBF);
   uint16_t my_crc16_t=crc16wtable(0,0,(0x020000-2),MSBF);

   while(1)
   {
   }
}
```

8.10.3 CRC-32

Cerință: Să se realizeze o aplicație care **calculează și verifică** valoarea **CRC-32** a unui bloc de date stocat în memoria **flash**. Blocul de date are dimensiunea de **128 KB** și este plasat la adresa **0x000000–0x01FFFF**, iar valoarea **CRC-32** "reală" este salvată imediat după acesta, la adresa **0x020000–0x020003**. Aplicația va calcula CRC-ul folosind **două metode diferite**:

1. Metoda "**bit-cu-bit**" (fără tabelă);
2. Metoda cu **tabele precalculate** (lookup table).

`crc32.h`

`crc32.h` reprezintă header-ul modulului de calcul al valorii **CRC pe 32 biți**, declarând funcțiile necesare pentru: calculul **CRC-32** prin metoda clasică **bit-cu-bit** (**`crc32_f()`**), folosind un polinom specificat și o valoare inițială, calculul **CRC-32** prin metoda cu **tabelă precalculată** (**`crc32wtable()`**), optimizată pentru performanță, selectarea ordinii de procesare a biților (**`enum BitOrder`** – **LSBF** sau **MSBF**), utilizarea polinoamelor standard definite pentru CRC-16 în reprezentare MSBF (**`0x4c11db7`**) și LSBF (**`0xEDB88320`**).

```
/*---------------------------------------------------------------------------
 * Fișier: crc32.h
 * Utilizat pentru declararea funcțiilor care calculează CRC-32
 *-------------------------------------------------------------------------*/

#ifndef _CRC_H_
#define _CRC_H_

/*---------------------------------------------------------------------------
 * Includes
 *-------------------------------------------------------------------------*/

// General
#include <ioavr.h>
#include <inavr.h>
#include <stdint.h>

/*---------------------------------------------------------------------------
 * Data structures
 *-------------------------------------------------------------------------*/

/*
 * Enum care definește ordinea de procesare a biților.
 * LSBF: bitul cel mai puțin semnificativ este procesat primul.
 * MSBF: bitul cel mai semnificativ este procesat primul.
 */
enum BitOrder {LSBF, MSBF};

/*---------------------------------------------------------------------------
 * Public defines
 *-------------------------------------------------------------------------*/

// Polinomul standard pentru CRC-32 în reprezentare MSBF
#define CRC32_MSBF 0x4c11db7

// Polinomul standard pentru CRC-32 în reprezentare LSBF
#define CRC32_LSBF 0xEDB88320
```

```
/*--------------------------------------------------------------------------
 * Public (exported) functions
 *-------------------------------------------------------------------------*/

/*
 * Funcția calculează CRC-32 folosind o tabelă pre-calculată (lookup table)
 * și returnează valoarea finală calculată.
 */
uint32_t crc32_f(uint32_t polinom32, uint32_t init_val_32,
                 uint32_t adr_start, uint32_t len, enum BitOrder ord);

/*
 * Funcția calculează CRC-32 folosind metoda clasică bit-cu-bit (fără
 * tabelă) și returnează valoarea finală calculată.
 */
uint32_t crc32wtable(uint32_t init_val_32, uint32_t adr_start,
                     uint32_t len, enum BitOrder ord);

#endif
```

crc32.c

Fișierul **crc32.c** definește funcțiile și tabelele necesare pentru calculul **CRC-32**, atât pentru metoda optimizată cu **tabele precalculate** (lookup table) cât și pentru ambele ordini de procesare a biților: **LSB-first** și **MSB-first.** Acesta conține 2 tabele în memoria flash (**crc32_tab_LSBF** și **crc32_tab_MSBF**) care permit calculul rapid al **CRC-ului** prin reducerea numărului de operații la fiecare octet procesat. Valorile din tabele sunt generate în prealabil pe baza **polinomului standard** pentru **CRC-32**, astfel încât procesarea se rezumă la operații de tip **indexare** și **XOR**. Implementarea permite alegerea modului de procesare a datelor (**LSB-first** sau **MSB-first**) în funcție de cerințele aplicației, oferind suport pentru detectarea și verificarea erorilor în transmisia sau stocarea datelor. Utilizarea tabelelor precalculate asigură performanță superioară față de metodele clasice **bit-cu-bit**, fiind potrivită pentru aplicații unde viteza de calcul este critică.

```
/*--------------------------------------------------------------------------
 * Fișier: crc32.c
 * Utilizat pentru definirea funcțiilor și a tablourilor folosite de CRC-32
 *-------------------------------------------------------------------------*/

/*--------------------------------------------------------------------------
 * Includes
 *-------------------------------------------------------------------------*/

// General
#include "crc32.h"

/*--------------------------------------------------------------------------
 * Public variables
 *-------------------------------------------------------------------------*/

/*
 * Look-up table cu valori precalculate pentru optimizarea calculului CRC-32
 * cu LSB-first
 */
__flash unsigned long crc32_tab_LSBF[256] = {
0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3,
0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988, 0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5,
0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172, 0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924, 0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d,
0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e, 0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457,
```

```
0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb,
0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0, 0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9,
0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad,
0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a, 0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683,
0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb, 0x196c3671, 0x6e6b06e7,
0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc, 0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79,
0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236, 0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f,
0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38, 0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21,
0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db,
0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf,
0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94, 0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
};

//MSBF look-up table
__flash unsigned long crc32_tab_MSBF[256] = {
0x00000000, 0x04c11db7, 0x09823b6e, 0x0d4326d9, 0x130476dc, 0x17c56b6b, 0x1a864db2, 0x1e475005,
0x2608edb8, 0x22c9f00f, 0x2f8ad6d6, 0x2b4bcb61, 0x350c9b64, 0x31cd86d3, 0x3c8ea00a, 0x384fbdbd,
0x4c11db70, 0x48d0c6c7, 0x4593e01e, 0x4152fda9, 0x5f15adac, 0x5bd4b01b, 0x569796c2, 0x52568b75,
0x6a1936c8, 0x6ed82b7f, 0x639b0da6, 0x675a1011, 0x791d4014, 0x7ddc5da3, 0x709f7b7a, 0x745e66cd,
0x9823b6e0, 0x9ce2ab57, 0x91a18d8e, 0x95609039, 0x8b27c03c, 0x8fe6dd8b, 0x82a5fb52, 0x8664e6e5,
0xbe2b5b58, 0xbaea46ef, 0xb7a96036, 0xb3687d81, 0xad2f2d84, 0xa9ee3033, 0xa4ad16ea, 0xa06c0b5d,
0xd4326d90, 0xd0f37027, 0xddb056fe, 0xd9714b49, 0xc7361b4c, 0xc3f706fb, 0xceb42022, 0xca753d95,
0xf23a8028, 0xf6fb9d9f, 0xfbb8bb46, 0xff79a6f1, 0xe13ef6f4, 0xe5ffeb43, 0xe8bccd9a, 0xec7dd02d,
0x34867077, 0x30476dc0, 0x3d044b19, 0x39c556ae, 0x278206ab, 0x23431b1c, 0x2e003dc5, 0x2ac12072,
0x128e9dcf, 0x164f8078, 0x1b0ca6a1, 0x1fcdbb16, 0x018aeb13, 0x054bf6a4, 0x0808d07d, 0x0cc9cdca,
0x7897ab07, 0x7c56b6b0, 0x71159069, 0x75d48dde, 0x6b93dddb, 0x6f52c06c, 0x6211e6b5, 0x66d0fb02,
0x5e9f46bf, 0x5a5e5b08, 0x571d7dd1, 0x53dc6066, 0x4d9b3063, 0x495a2dd4, 0x44190b0d, 0x40d816ba,
0xaca5c697, 0xa864db20, 0xa527fdf9, 0xa1e6e04e, 0xbfa1b04b, 0xbb60adfc, 0xb6238b25, 0xb2e29692,
0x8aad2b2f, 0x8e6c3698, 0x832f1041, 0x87ee0df6, 0x99a95df3, 0x9d684044, 0x902b669d, 0x94ea7b2a,
0xe0b41de7, 0xe4750050, 0xe9362689, 0xedf73b3e, 0xf3b06b3b, 0xf771768c, 0xfa325055, 0xfef34de2,
0xc6bcf05f, 0xc27dede8, 0xcf3ecb31, 0xcbffd686, 0xd5b88683, 0xd1799b34, 0xdc3abded, 0xd8fba05a,
0x690ce0ee, 0x6dcdfd59, 0x608edb80, 0x644fc637, 0x7a089632, 0x7ec98b85, 0x738aad5c, 0x774bb0eb,
0x4f040d56, 0x4bc510e1, 0x46863638, 0x42472b8f, 0x5c007b8a, 0x58c1663d, 0x558240e4, 0x51435d53,
0x251d3b9e, 0x21dc2629, 0x2c9f00f0, 0x285e1d47, 0x36194d42, 0x32d850f5, 0x3f9b762c, 0x3b5a6b9b,
0x0315d626, 0x07d4cb91, 0x0a97ed48, 0x0e56f0ff, 0x1011a0fa, 0x14d0bd4d, 0x19939b94, 0x1d528623,
0xf12f560e, 0xf5ee4bb9, 0xf8ad6d60, 0xfc6c70d7, 0xe22b20d2, 0xe6ea3d65, 0xeba91bbc, 0xef68060b,
0xd727bbb6, 0xd3e6a601, 0xdea580d8, 0xda649d6f, 0xc423cd6a, 0xc0e2d0dd, 0xcda1f604, 0xc960ebb3,
0xbd3e8d7e, 0xb9ff90c9, 0xb4bcb610, 0xb07daba7, 0xae3afba2, 0xaafbe615, 0xa7b8c0cc, 0xa379dd7b,
0x9b3660c6, 0x9ff77d71, 0x92b45ba8, 0x9675461f, 0x8832161a, 0x8cf30bad, 0x81b02d74, 0x857130c3,
0x5d8a9099, 0x594b8d2e, 0x5408abf7, 0x50c9b640, 0x4e8ee645, 0x4a4ffbf2, 0x470cdd2b, 0x43cdc09c,
0x7b827d21, 0x7f436096, 0x7200464f, 0x76c15bf8, 0x68860bfd, 0x6c47164a, 0x61043093, 0x65c52d24,
0x119b4be9, 0x155a565e, 0x18197087, 0x1cd86d30, 0x029f3d35, 0x065e2082, 0x0b1d065b, 0x0fdc1bec,
0x3793a651, 0x3352bbe6, 0x3e119d3f, 0x3ad08088, 0x2497d08d, 0x2056cd3a, 0x2d15ebe3, 0x29d4f654,
0xc5a92679, 0xc1683bce, 0xcc2b1d17, 0xc8ea00a0, 0xd6ad50a5, 0xd26c4d12, 0xdf2f6bcb, 0xdbee767c,
0xe3a1cbc1, 0xe760d676, 0xea23f0af, 0xeee2ed18, 0xf0a5bd1d, 0xf464a0aa, 0xf9278673, 0xfde69bc4,
0x89b8fd09, 0x8d79e0be, 0x803ac667, 0x84fbdbd0, 0x9abc8bd5, 0x9e7d9662, 0x933eb0bb, 0x97ffad0c,
0xafb010b1, 0xab710d06, 0xa6322bdf, 0xa2f33668, 0xbcb4666d, 0xb8757bda, 0xb5365d03, 0xb1f740b4
};

/*-------------------------------------------------------------------------
 * Public functions
 *-----------------------------------------------------------------------*/

/*
 * Funcția calculează CRC-32 folosind metoda clasică bit-cu-bit (fără
 * tabelă) și returnează valoarea finală calculată.
 */
uint32_t crc32_f(uint32_t polinom32, uint32_t init_val_32,
                 uint32_t adr_start, uint32_t len, enum BitOrder ord)
{
  uint32_t crc = init_val_32; // Inițializează CRC cu valoarea de start
  uint32_t data = 0;          // Variabilă temporară pentru octetul curent
  while(len--) {
    int i;
    // Se extrage valoarea octetului de la adresa de start din memoria flash
    data = *(__farflash char *)adr_start;
    if ( ord == MSBF ) // Varianta cu shiftare spre MSB
```

```
    {
      data <<= 24; // Se aliniază octetul la MSB
      crc ^= data; // Se transferă și integrează datele în CRC
      adr_start++;
      for(i = 0; i < 8; ++i) {
        if( crc & 0x80000000) // Se verifică dacă MSB este 1
          crc = (crc << 1) ^ polinom32;
        else
          crc = crc << 1;
      }
    }
    else { // Varianta cu shiftare spre LSB
      crc ^= data;
      adr_start++;
      for(i = 0; i < 8; ++i) { // Se verifică dacă LSB este 1
        if(crc & 0x00000001)
          crc = (crc >> 1) ^ polinom32;
        else
          crc = crc >> 1;
      }
    }
  }
  return crc;
}

/*
 * Funcția calculează CRC-16 folosind o tabelă pre-calculată (lookup table)
 * și returnează valoarea finală calculată.
 */
uint32_t crc32wtable(uint32_t init_val_32, uint32_t adr_start,
                     uint32_t len, enum BitOrder ord)
{

  uint32_t counter;
  uint32_t crc = init_val_32; // Inițializează CRC cu valoarea de start
  for( counter = 0; counter < len; counter++)
  {
    if(ord == MSBF) // Varianta cu MSB
    /* 1. Shift la stânga CRC cu 8 biți
     * 2. XOR cu valoarea din tabelul MSBF corespunzătoare.
     * Indexul în tabel se obține astfel:
     * - (crc >> 24) = octetul superior al CRC curent
     * - XOR cu byte-ul citit din memorie (de la adr_start)
     * 3. Se adună valoarea din tabel, care reprezintă efectul acelui byte
     * asupra CRC-ului.
     */
      crc = (crc << 8) ^ crc32_tab_MSBF[((crc >> 24) ^ *(__farflash char
            *)adr_start++) & 0xff];
    else // Varianta cu LSB
      crc = (crc >> 8) ^ crc32_tab_LSBF[(crc ^ *(__farflash char
            *)adr_start++) & 0xff];
  }
    return crc;
}
```

`main.c`

Fișierul **`main.c`** reprezintă punctul de intrare al aplicației **CRC-32**. În cadrul acestuia, se citește valoarea reală a CRC-ului stocată în memoria flash (**`real_crc`**), apoi se calculează **CRC-ul** pentru aceeași zonă de memorie prin două metode diferite:

- **`crc16()`** – metoda clasică **bit-cu-bit**, care procesează fiecare bit în funcție de polinomul specificat.
- **`crc16wtable()`** – metoda optimizată cu **tabele precalculate** (look-up table), care reduce numărul de operații necesare pentru calculul CRC-ului.

Ambele metode procesează datele din memoria flash, cu ordinea biților specificată (**MSB-first**). Programul nu conține o buclă de execuție funcțională, rămânând blocat în bucla infinită **`while(1)`** după efectuarea calculelor, permițând astfel analizarea și compararea rezultatelor obținute prin cele două tehnici.

```
/*------------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul principial de rulare a aplicației CRC-32
 *----------------------------------------------------------------------*/

/*------------------------------------------------------------------------
 * Includes
 *----------------------------------------------------------------------*/

// General
#include "crc32.h"

void main(void) {
/*
 * Se calculează atât CRC-ul folosind funcția standard crc16(), cât și cu
 * crc16wtable(), versiunea cu look-up table.
 */
   uint32_t real_crc = *(__farflash uint32_t *)(0x20000 - 4);
   uint32_t my_crc32 = crc32_f(CRC32_MSBF,0, 0, (0x20000 - 4), MSBF);
   uint32_t my_crc32_t = crc32wtable(0, 0, (0x20000 - 4), MSBF);
while(1) {
   }
}
```

8.11 BIBLIOGRAFIE

1. *"8-bit AVR Microcontroller ATmega 1280 Datasheet", Microchip Technology*
2. *"Schematic for UNI Clicker", Mikroe*
3. *"Schematic for ATmega1280: SiBrain", Mikroe*
4. *"Cyclic Redundancy Check", Wikipedia*

9. SERIAL PERIPHERAL INTERFACE – SPI

9.1 UNDE SE FOLOSEȘTE SPI?

SPI este util într-o gamă largă de produse prezente în viața de zi cu zi, fiind în principal folosit pentru transmiterea datelor între componente:

- **Touchscreen:** transmiterea datelor de la touchscreen controller la procesor;
- **Camere foto:** transmiterea informațiilor între obiectiv și camera foto;
- **Display LCD:** transmiterea rapidă a datelor permite afișarea imaginilor și a textului în timp real;
- **Carduri SD:** transmiterea datelor în ambele sensuri între card și diferite dispozitive.

Figura 9.1 – Un display LCD cu SPI (stânga) și un modul de cameră foto SPI (dreapta)

9.2 CUNOȘTINȚE NECESARE

Pentru a putea parcurge acest capitol, este necesar să cunoașteți următoarele subiecte din capitolele anterioare, și nu numai:

- Arhitectura Master-Slave;
- Utilizarea întreruperilor;
- Utilizarea kit-ului hardware Mikroe;
- Comunicarea sincronă.

9.3 ABSTRACT

Acest capitol are în vedere detalierea protocolului de comunicare **SPI** (**Serial Peripheral Interface**) împreună cu block-ul hardware dedicat prezent în structura microcontrolerului ATmega 1280. Pentru a exemplifica modul de funcționare al protocolului și al blocului hardware, s-a propus:

- desenarea pe un ecran LCD a unor forme geometrice;
- folosirea matricii de led-uri 7x10.

9.4 HARDWARE ȘI SOFTWARE NECESAR

- ATmega 1280 SiBRAIN;
- UNI Clicker;
- Atmel ICE;
- Matrice LED 7x10;
- Osciloscop;
- IAR Embedded Workbench 7.30.5.

9.5 MATRICEA LED 7X10B CLICK

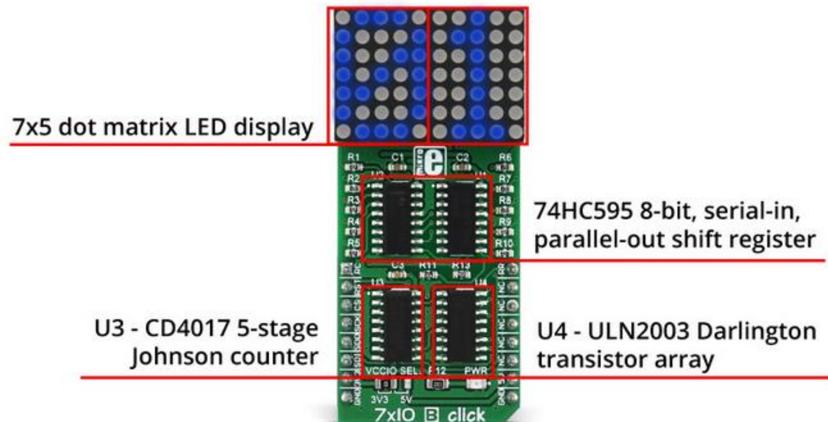


Figura 9.2 – Modulul 7x5 LED Dot Matrix controlat de 74HC595, CD4017 și ULN2003

9.5.1 FUNCȚIONALITATE

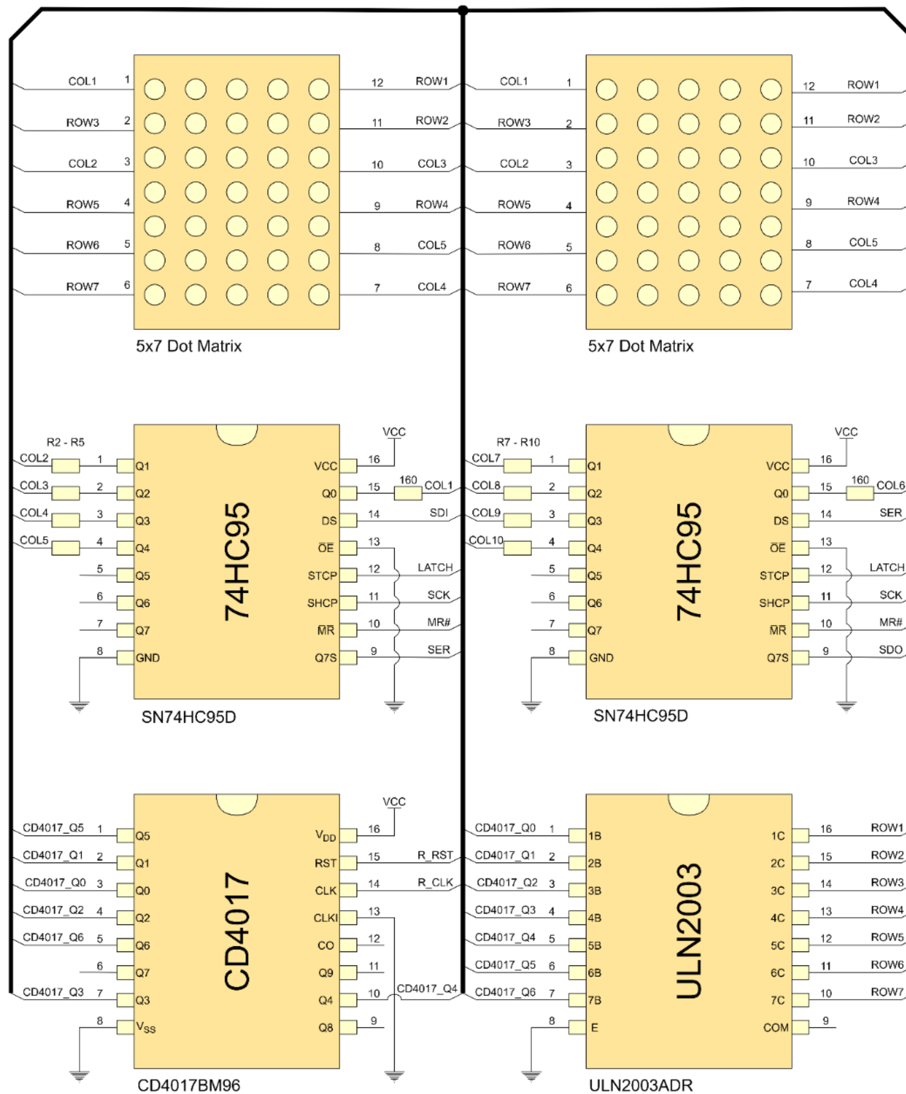


Figura 9.3 – Schema click-ului 7x10

9.5.1.1 7X5 DOT MATRIX LED DISPLAY

Display-ul **7x5 Dot Matrix** este compus din **35** de **LED-uri** organizate într-o rețea de **7 rânduri** (anod comun) și **5 coloane** (catod comun). În figura de mai sus, au fost combinate **2 matrici** pentru posibilitatea afișării unui contor de la **0 la 99**. Pentru a reduce consumul de pini și complexitatea cablajului, afișajul este controlat prin multiplexare pe rânduri: la un moment dat este activ un singur rând, dar schimbarea rapidă între rânduri dă iluzia afișării complete.

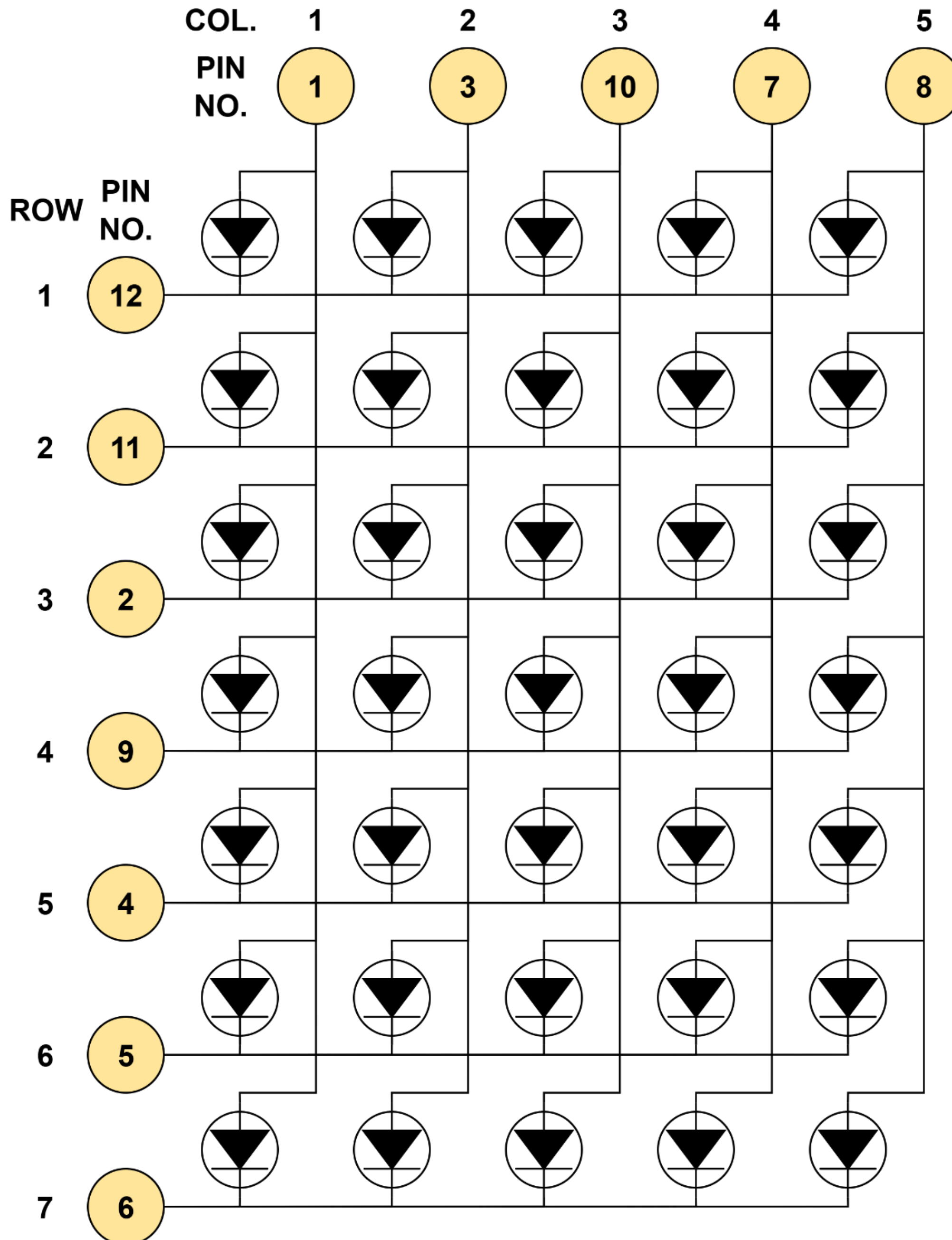


Figura 9.4 – Schema electrică a matricei LED

Controlul afișajului se face cu ajutorul componentelor și procedurilor enumerate mai jos.

9.5.1.2 REGISTRUL DE DEPLASARE 74HC595

Sunt utilizate unul sau două circuite **74HC595** (în funcție de lățimea matricei) pentru a controla coloanele LED-urilor. Fiecare registru **primește** date seriale de la microcontroler prin **interfața SPI** (sau similar), bit cu bit. După ce datele sunt încărcate în registre (ex: 2 octeți pentru o matrice 7x10), un semnal de **latch** (**STCP**) transferă

simultan datele în ieșirile paralele **Q0–Q7**. Aceste ieșiri polarizează bornele pozitive (anozi) ale coloanelor. Însă LED-urile nu se aprind până când și rândul corespunzător este activat.

9.5.1.3 MULTIPLEXAREA RÂNDURILOR CU CD4017 (JOHNSON COUNTER)

CD4017 este un contor decade care avansează câte o ieșire activă la fiecare impuls de clock. În aplicația noastră, doar primele 7 ieșiri sunt folosite pentru a selecta secvențial rândurile matricei. La fiecare impuls de ceas, contorul **activează un singur rând** (anod comun), indicând că datele pentru acel rând sunt gata să fie afișate. După activarea ultimului rând, contorul **se resetează** și ciclul începe din nou.

CLK	$\overline{CE}$	MR	OUTPUT STATE †
LOW	X	LOW	Nu apare nicio schimbare
X	HIGH	LOW	Nu apare nicio schimbare
X	X	HIGH	“0” = HIGH, “1” – “9” = LOW
URCĂ ↑	LOW	LOW	Incrementează counter-ul
COBOARĂ ↓	X	LOW	Nu apare nicio schimbare
X	URCĂ ↑	LOW	Nu apare nicio schimbare
HIGH	COBOARĂ ↓	LOW	Incrementează counter-ul

Tabelul 9.1 – Tabela de adevăr pentru counter-ul Johnson CD4017
† - dacă n < 5 TC (Terminal Count) = HIGH, altfel TC = LOW

9.5.1.4 COMUTAREA DE PUTERE – ULN2003A

Pentru a comuta curentul necesar aprinderii **LED-urilor**, ieșirile **CD4017** sunt conectate la un driver de curent tip **Darlington – ULN2003A**. Acest circuit primește semnalul de la **CD4017** și conectează rândul activ la masă (**GND**). Astfel, doar LED-urile ale căror coloane sunt polarizate **pozitiv** (via **74HC595**) și al căror rând este activ (prin **ULN2003A**) vor conduce curentul și se vor aprinde. **ULN2003** oferă și diode de protecție interne împotriva tensiunilor inverse, protejând componentele logice.

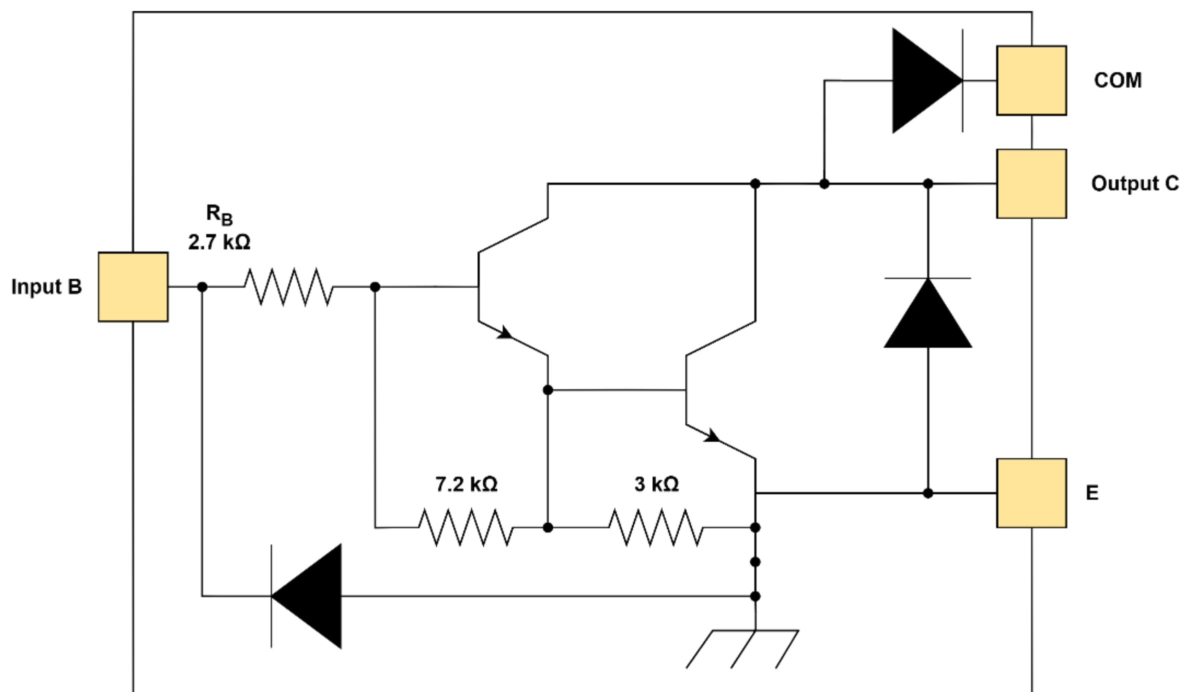


Figura 9.5 – Schema electrică a lui ULN2003

9.5.1.5 ACTUALIZARE CICLICĂ

Întregul proces de afișare constă în următoarele etape repetitive:
- Microcontrolerul transmite pattern-ul de date pentru coloane către **74HC595**;
- Se generează un semnal de **latch** care aplică datele la ieșire;
- Se activează unul dintre cele 7 rânduri (prin **CD4017** și **ULN2003A**);
- Se menține starea un timp foarte scurt (**~1–2 ms**);
- Se trece la următorul rând și **ciclul se repetă**.

Acest ciclu trebuie să se repete la o frecvență minimă de **~100 Hz** pentru ca ochiul uman să perceapă o imagine stabilă, continuă. Acesta este principiul de bază al afișajului multiplexat.

9.5.2 REZULTATE

Prin această arhitectură:
- se pot afișa caractere **ASCII**, **cifre**, **simboluri** și **animații scurte** (ex: scrolling text);
- se economisesc pini **GPIO** ai microcontrolerului (doar 3–4 pini sunt necesari);
- sistemul este **modular și scalabil**, fiind potrivit pentru aplicații **embedded** cu microcontrolere cu resurse limitate (ex: Arduino, STM32, PIC, etc.).

Controlul se realizează folosind 3 pini principali:
- **Data** (pentru **74HC595**);
- **Clock** (pentru **74HC595** și **CD4017**);
- **Latch** (pentru **74HC595**).

9.6 INTRODUCERE ÎN SPI

Definiție

SPI (Serial Peripheral Interface) este un standard de transmisie serial sincron de tip Master-Slave, full duplex, folosit pentru a permite comunicarea de viteză mare între dispozitivele electronice integrate.

Acest standard a fost inventat la începutul anilor 1980 de către **Motorola** și presupune ca bus-ul să aibă, pe lângă linia de **clock** și **liniile de date**, câte o linie de la **Master** la fiecare dispozitiv **Slave** din rețea pentru inițierea transmisiei.

Comunicarea **SPI** se bazează pe o arhitectură **Master-Slave**. Comunicarea este inițiată și controlată de **Master**, care selectează un dispozitiv **Slave** prin coborârea liniei **SS** (**Slave Select**) la nivel logic **LOW** (0 logic). Pentru sincronizarea schimbului de date, **Master-ul** generează semnalul de ceas pe linia **SCK** / **SCLK** (**Serial Clock**). Protocolul **SPI** permite transferul de date simultan în ambele direcții, datorită funcționării sale în mod full-duplex:
- **MOSI** (**Master Out**, **Slave In**): **Master-ul** trimite date către **Slave**.
- **MISO** (**Master In**, **Slave Out**): **Slave-ul** trimite date către **Master**.

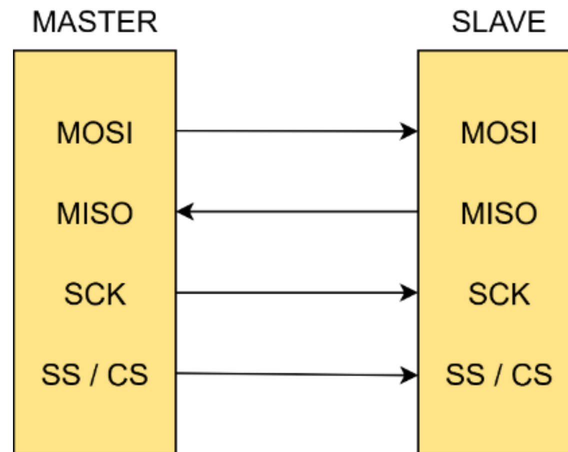


Figura 9.6 – Comunicarea SPI între Master și Slave

SPI este un standard flexibil, având mai multe variații ale modului de comunicare cu dispozitivele precum:

- Lipsa semnalului **Slave Select** / **Chip Select**;
- Variații de timing prin modificarea fazei și polarității clock-ului utilizând biții **CPOL** (**Clock POLarity**) și **CPHA** (**Clock PHAse**);
- Dimensiunea unui **WORD** (de obicei **8 biți**);
- Schimbarea modului de utilizare a conectorilor (de exemplu conectorii standardului **JTAG**);
- Apariția semnalului **Flow Control** (**RDY**) folosit pentru a semnala **Master-ului** că **Slave-ul** este pregătit de transmis date.

9.7 INTERFAȚA SPI

Standardul propus de Motorola (acum deținut de **NXP**) folosește 4 semnale unidirecționale:

- **CS** / **SS** – **Chip** / **Slave Select**: Este un semnal de obicei activ pe **LOW** folosit de **Master** pentru a iniția comunicarea cu **Slave-ul**;
- **SCK** – **Serial ClocK**: Clock-ul comun impus de **Master**;
- **MOSI** – **Master In Slave Out**: Pin pentru datele care vin de la **Master** la **Slave**;
- **MISO** – **Master Out Slave In**: Pin pentru datele care vin de la **Slave** la **Master**.

Notă: În documentațiile noi termenii **Master** și **Slave** sunt înlocuiți cu ***Controller*** și ***Peripheral***.

9.7.1 TRANSMISIA DE DATE ÎN MODUL MASTER

Pentru a începe o transmisie, **Master-ul** trebuie să activeze semnalul **SS** corespunzător **Slave-ului** după care să trimită impulsurile de clock. Pentru fiecare impuls de clock **se transmite** și / sau **se primește** un bit. După terminarea trasnsmisiei **se dezactivează** semnalul **SS** al **Slave-ului**.

9.7.2 TRANSMISIA DE DATE ÎN MODUL SLAVE

Atât timp cât linia **SS** este ținută pe **HIGH**, **Slave-ul** nu poate să comunice cu **Master-ul**, liniile de date fiind deconectate (tri-stated). Dacă linia **SS** este ținută **LOW**, **Slave-ul** primește și transmite datele după clock-ul impus de **Master**. **SS** sincronizează generatorul de clock al **Master-ului** cu registrul de shiftare al **Slave-ului**, asigurând transmisia corectă a pachetelor de date.

9.7.3 TOPOLOGII DE REȚELE DE COMUNICARE

9.7.3.1 MULTIDROP

Această configurare este cea mai utilizată topologie de bus pentru protocolul **SPI**. **Multidrop** presupune ca **Master-ul** să utilizeze câte un pin **I/O** dedicat conectat la **SS-ul** fiecărui **Slave** din rețea, lucru ce permite

transmisia de date la mai mulți **Slave** în același timp (aceleași date **1-N**). Recepția de date de la **Slave-uri** va rămâne **1-1**.

9.7.3.2 DAISY CHAIN

Definiție

Daisy Chain-ul este format conectând ieșirea unui Slave la intrarea următorului Slave din șir. Master-ul va avea intrarea conectată la ultimul Slave din lanț, iar ieșirea la primul Slave.

Această configurație folosește doar un singur pin **SS**, dar crește durata transmiterii datelor între dispozitivele rețelii.

9.7.3.3 EXPANDER

Definiție

Expander-ul este o componentă folosită pentru a crește numărul de Slave cu care poate comunica Master-ul.

De exemplu, folosirea unui demultiplexor va crește numărul de la **n** la **2n** utilizând **n + 1** pini **I/O** (**+1** este pentru activarea demultiplexorului, altfel comunicarea se va face în permanență), unde se vor folosi **n** linii pentru a alege un **Slave**.

9.8 PERIFERICUL SPI

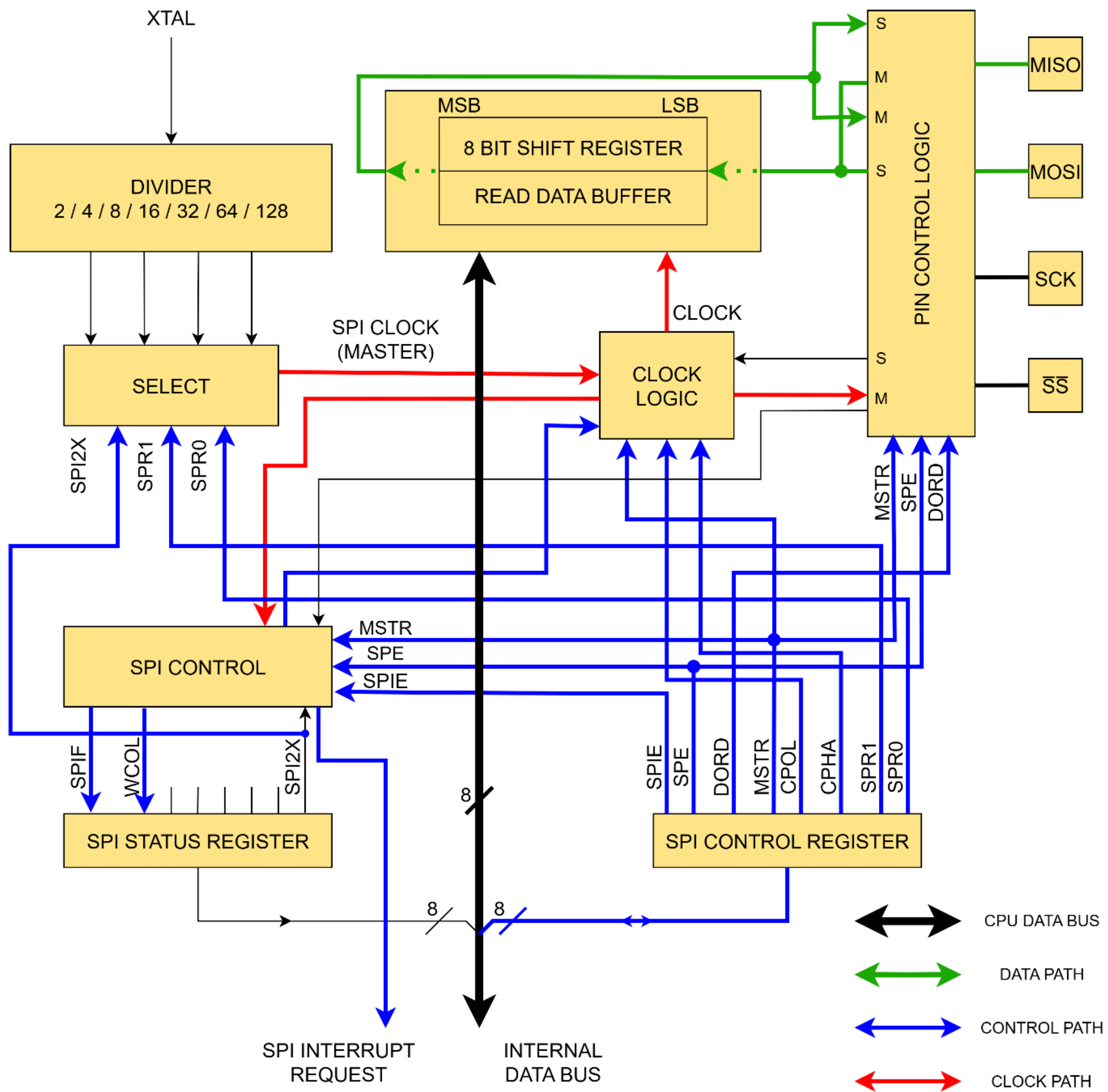


Figura 9.7 - Diagrama bloc a modulului SPI

9.8.1 MODURI DE OPERARE

Dispozitivul poate fi configurat ca **Master** (inițiază transferul de date, controlează semnalul de clock, activează dispozitivul **Slave** prin pinul **SS** – **Slave Select**).

De asemenea, dispozitivul poate fi configurat ca **Slave** (primește semnalul **SCK** de la **Master** folosit pentru sincronizarea citirii / scrierii de date, primește date pe **MOSI** și le trimite pe **MISO**, doar dacă este interogat de **Master**).

9.8.2 MOD DE PROGRAMARE

9.8.2.1 CONFIGURAREA

Pentru a comunica prin **SPI**, se pornește modulul **SPI** din registrul de control, se configurează microcontrolerul ca **Master** (cel care controlează comunicarea), se alege cât de repede se face schimbul de date (de obicei ca fracțiune din frecvența ceasului intern), se decide dacă transferul va fi **LSB** sau **MSB first** și se setează polaritatea și faza semnalului de clock, în funcție de cerințele dispozitivului conectat.

9.8.2.2 SELECTAREA

Master-ul alege un **Slave** cu care să comunice setând pinul **SS** în starea **LOW**.

9.8.2.3 INIȚIEREA TRANSFERULUI

Se scrie octetul de date destinat transmisiei în registrul **SPDR**. Această acțiune încarcă octetul în registrul de deplasare și declanșează pornirea transferului. După pornirea transferului, **Master-ul** începe să genereze impulsuri de **SCK**. Frecvența **SCK** este determinată de **CLK** și este configurată printr-un prescaler. Impulsurile nu sunt generate continuu, ci doar în timpul transferului. Între transferuri, **SCK** rămâne fie **HIGH**, fie **LOW**, în funcție de bitul **CPOL**. Dacă **CPOL** este **0**, **SCK** este **LOW** în stare de repaus. Dacă în schimb **CPOL** este **1**, **SCK** este **HIGH** în stare de repaus.

9.8.2.4 SCHIMBUL DE DATE

La fiecare impuls de ceas, are loc un schimb de date simultan (**full-duplex**):

- **Master-ul** trimite un bit pe **MOSI**.
- **Slave-ul** trimite un bit pe **MISO**.

9.8.2.5 FINALIZAREA TRANSFERULUI

După 8 cicluri de ceas, transferul este complet. Octetul recepționat este transferat din registrul de deplasare în buffer-ul de citire, iar flag-ul **SPIF** (flagul de transmisie) din registrul de status (**SPSR**) este setat.

9.8.2.6 SINCRONIZAREA ȘI CITIREA

Software-ul detectează **SPIF = 1** sau o întrerupere. Apoi, acesta poate citi octetul recepționat din **SPDR**.

9.8.2.7 DEZACTIVAREA SLAVE-ULUI

Pinul **SS** este comutat în starea inactivă pentru a finaliza comunicarea cu dispozitivul **Slave**.

9.9 COMPONENTE

9.9.1 PINII MODULULUI SPI

Activarea modulului **SPI** al microcontrolerului va configura următorii pini ca interfață a modulului:

- **MISO - PB3**
- **MOSI - PB2**
- **SCK - PB1**
- **SS - PB0**

Pinul **SS** este comutat în starea inactivă pentru a finaliza comunicarea cu dispozitivul Slave.

9.9.2 REGIȘTRII MODULULUI SPI

9.9.2.1 SPCR - REGISTRUL DE CONTROL SPI

Bit	7	6	5	4	3	2	1	0	
0x2C (0x4C)	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
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 9.8 - Registrul de control SPI

Aceasta este componenta care programează cum va funcționa modulul **SPI**:

- **SPIE** (**SPI Interrupt Enable**): Activează întreruperea **SPI**. Când un transfer se termină, se va genera o cerere de întrerupere.
- **SPE** (**SPI Enable**): Activează sau dezactivează complet modulul **SPI**.
- **DORD** (**Data Order**): Stabilește ordinea biților. Se pot trimite date fie **MSB** (**Most Significant Bit**) primul, fie **LSB** (**Least Significant Bit**) first.
- **MSTR** (**Master / Slave Select**): Setează modul de operare: **1** pentru **Master**, **0** pentru **Slave**.
- **CPOL / CPHA** (**Clock Polarity / Phase**): Definesc cele 4 moduri de funcționare **SPI**, stabilind polaritatea ceasului în repaus și pe ce front (crescător / descrescător) se eșantionează datele.

	Frontul Principal	Frontul Secundar	SPI Mode
CPOL = 0, CPHA = 0	Devine HIGH	Devine LOW	0
CPOL = 0, CPHA = 1	Devine LOW	Devine HIGH	1
CPOL = 1, CPHA = 0	Devine HIGH	Devine LOW	2
CPOL = 1, CPHA = 1	Devine LOW	Devine HIGH	3

Tabelul 9.2 - Cele 4 moduri de funcționare SPI în funcție de CPOL / CPHA

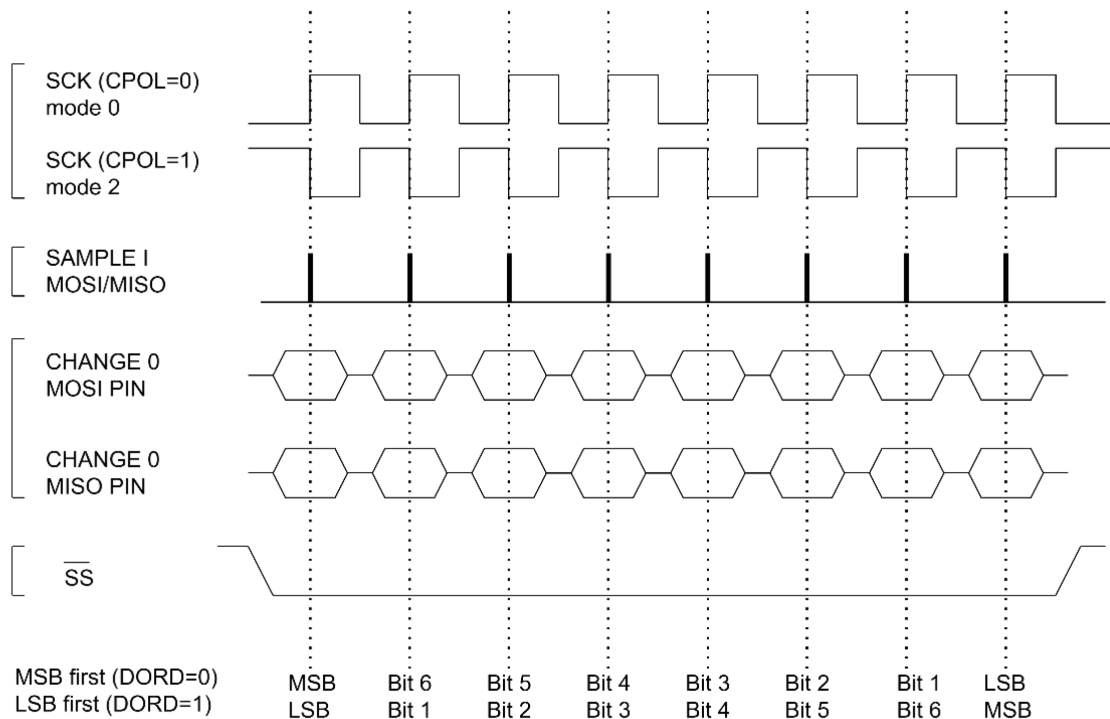


Figura 9.9 - SPI Transfer Format cu CPHA = 0

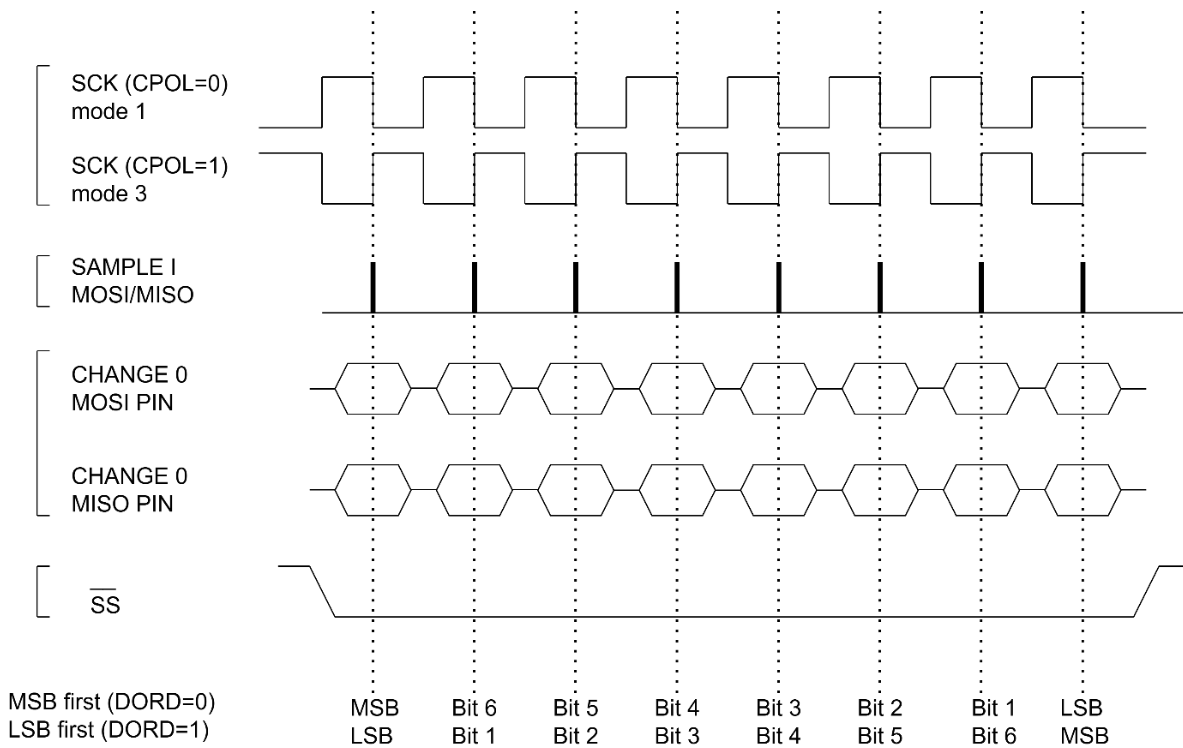


Figura 9.10 - SPI Transfer Format cu CPHA = 1

- **SPR1**, **SPR0** (**SPI Clock Rate Select**): Acești biți, împreună cu bitul **SPI2X**, selectează viteza ceasului **SCK** atunci când dispozitivul este în modul **Master**.

SPI2X	SPR1	SPR0	Frecvența SCK
0	0	0	f_{osc} / 4
0	0	1	f_{osc} / 16
0	1	0	f_{osc} / 64
0	1	1	f_{osc} / 128
1	0	0	f_{osc} / 2
1	0	1	f_{osc} / 8
1	1	0	f_{osc} / 32
1	1	1	f_{osc} / 64

Tabelul 9.3 - Frecvența SCK în funcție de SPI2X, SPR1 și SPR0 (unde f_{osc} este implicit 8 MHz)

9.9.2.2 SPSR - REGISTRUL DE STARE SPI

Acesta reține informații despre starea curentă a modulului.

Bit	7	6	5	4	3	2	1	0	
0x2D (0x4D)	SPIF	WCOL	-	-	-	-	-	SPI2X	SPSR
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 9.11 - Registrul de stare SPSR

- **Bitul 7 – SPIF** (**SPI Interrupt Flag**)
  După un transfer complet, acest bit se setează automat. Dacă **SPIE** din **SPCR** este și el setat și întreruperile sunt active (**I** din **SREG** este **1**), se generează întreruperea **SPI Serial Transfer Complete**.

- **Bitul 6 – WCOL** (**Write Collision**)
  Se setează automat dacă se scrie în **SPDR** în decursul unui transfer de date. Bitul **WCOL** (și **SPIF**) este resetat atunci când se citește **SPSR** și se accesează **SPDR**.

- **Biții 5:1 – Rezervați (se vor citi mereu 0)**

- **Bitul 0 – SPI2X** (**Double SPI Speed Bit**)
  Dacă modul **Master** este activ, frecvența modulului **SPI** se dublează (viteza maximă de transfer devine jumătate din frecvența microcontrolerului).

Notă: Când modul **Slave** este activ, viteza maximă cu care poate primi date rămâne ¼ din frecvența microcontrolerului.

9.9.2.3 SPDR - REGISTRUL DE DATE SPI

Folosit pentru transferul dintre regiștrii de uz general și registrul de shiftare **SPI.**

Bit	7	6	5	4	3	2	1	0	
0x2E (0x4E)	MSB							LSB	SPSR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial value	X	X	X	X	X	X	X	X	

Figura 9.12 - Registrul de date SPDR

Generarea clock-ului:
- **XTAL**: Reprezintă ceasul principal al sistemului.
- **DIVIDER**: Acest bloc preia clock-ul sistemului și îl împarte la diverse valori pentru a crea mai multe frecvențe posibile pentru clock-ul **SPI**.
- **Blocul SELECT**: Acest multiplexor selectează una dintre frecvențele generate de divizor, pe baza setărilor din biții **SPR0**, **SPR1** (din **SPCR**) și **SPI2X** (din **SPSR**).
- **SPI CLOCK** (**MASTER**): Acesta este semnalul de clock final, generat intern, care va fi scos pe pinul **SCK**.

Logica de control și pini:
- **CLOCK LOGIC**: Acest bloc primește semnalul de clock (fie cel generat intern în modul **Master**, fie cel extern de la pinul **SCK** în modul **Slave**) și coordonează deplasarea biților în registrul de deplasare, conform setărilor **CPOL / CPHA**.
- **PIN CONTROL LOGIC**: Acest bloc gestionează direcția pinilor fizici în funcție de modul de operare (**Master / Slave**), setat de bitul **MSTR**.

Registru de deplasare pe 8 biți:

Când se trimit date, octetul este încărcat aici și apoi deplasat bit cu bit spre ieșirea **MOSI**. Simultan, biții care vin pe intrarea **MISO** sunt încărcați în acest registru.

Buffer de citire a datelor:

Acesta este un buffer intermediar. La finalul unui transfer de 8 biți, conținutul registrului de deplasare (adică datele primite) este copiat aici. Acest lucru permite să se citească datele recepționate în timp ce modulul **SPI** începe deja un nou transfer (**double-buffering**), crescând eficiența.

9.10 PROBLEME

Pentru aplicațiile **SPI**, se pun la dispoziție următoarele biblioteci (headere) și funcțiile aferente:

`spi.h`

Header-ul **`spi.h`** declară funcțiile necesare pentru comunicarea **SPI**, mai exact inițializarea modulului **SPI** (**`spi_init()`**) și transmiterea unui octet către **Slave** (**`spi_transmit()`**), permițând separarea clară a interfeței de implementare.

```
/*------------------------------------------------------------------------
 * Fișier: spi.h
 * Utilizat pentru declararea funcțiilor SPI
 *----------------------------------------------------------------------*/

#ifndef _SPI_H
#define _SPI_H

#include <ioavr.h>
#include <inavr.h>
#include <stdint.h>

/*------------------------------------------------------------------------
 * Public (exported) functions
 *----------------------------------------------------------------------*/

/*
 * Funcția configurează interfața SPI a microcontrolerului, permițând
 * trimiterea datelor serial către matricea de LED-uri.
 */
void spi_init(void);

/*
 * Funcția transmite un byte prin SPI (doar din Master către Slave) și
 * așteaptă finalizarea transmiterii înainte de a continua.
 */
void spi_transmit(uint8_t data);

#endif
```

`spi.c`

Fișierul de funcții **`spi.c`** conține implementarea funcțiilor declarate în **`spi.h`**, unde **`spi_init()`** configurează microcontrolerul ca **Master SPI** (activând modulul, setând pinii, viteza și direcția de transmitere), iar **`spi_transmit()`** trimite un byte și așteaptă blocant până la finalizarea transmisiei, controlând astfel trimiterea datelor către registrele de deplasare ale matricei.

```
/*------------------------------------------------------------------------
 * Fișier: spi.c
 * Utilizat pentru definirea funcțiilor specifice SPI
 *----------------------------------------------------------------------*/

#include "matrix.h"

/*------------------------------------------------------------------------
 * Public defines
 *----------------------------------------------------------------------*/
```

```
// Încarcă registrul SPI și începe transmisia
#define TRANSMIT(data) (SPDR = data)
/*
 * Funcția spi_init:
 * - Configurează pinii necesari pentru comunicarea SPI
 * - Activează modul SPI în regim Master
 * - Setează viteza SPI la fosc/16
 * - Activează transmiterea LSB-first
 */
void spi_init(void) {
  // Se setează pinul PB1 (SCK), PB2 (MOSI) și PB0 (SS) ca output
  DDRB |= (1 << PB1) | (1 << PB2) | (1 << PB0);
  // Activare SPI, Master mode, prescaler fosc/16, LSB-first
  SPCR = (1 << SPE) | (1 << MSTR) | (1 << SPR0) | (1 << DORD);
}

/*
 * Funcția SPI_Transmit:
 * - Transmite un byte pe magistrala SPI
 * - Așteaptă finalizarea transmisiei (SPIF = 1)
 */
void spi_transmit(uint8_t data) {
  TRANSMIT(data);
  while (!(SPSR & (1 << SPIF)));  // Așteaptă finalizarea transmisiei
}
```

`matrix.h`

`matrix.h` reprezintă header-ul pentru logica de afișare pe matricea **LED**, declarând funcțiile necesare pentru inițializarea pinilor de control (**`matrix_init()`**), trimiterea datelor prin **SPI** către registrele de deplasare (**`transmitere_SPI()`**), generarea de impulsuri pentru controlul rândurilor (**`pulse_row_clock()`**, **`pulse_row_rst()`**) și pentru actualizarea ieșirii (**`pulse_latch()`**).

```
/*-------------------------------------------------------------------------
 * Fișier: matrix.h
 * Utilizat pentru declararea funcțiilor pentru matricea de LED-uri
 *-----------------------------------------------------------------------*/

#ifndef _MATRIX_H
#define _MATRIX_H
#include <ioavr.h>
#include <inavr.h>
#include <stdint.h>
#define private static

/*-------------------------------------------------------------------------
 * Private (exported) functions
 *-----------------------------------------------------------------------*/

/*
 * Funcția configurează pinii care controlează rândurile și coloanele
 * din matricea LED și setează semnalul de latch pe HIGH.
 */
private void gpio_init(void);

/*-------------------------------------------------------------------------
 * Public (exported) functions
 *-----------------------------------------------------------------------*/

// Funcția activează modul SPI și setează pinii aferenți ca output.
void matrix_init(void);
```

```
/*
 * Funcția generează un puls pe pin-ul latch, trimițând datele către
 * ieșirile fizice ale registrului.
 */
void pulse_latch(void);

/*
 * Funcția generează un puls de reset pentru rândurile matriciei, aducând
 * pointerul de rând înapoi la început (pe prima linie).
 */
void pulse_row_rst(uint8_t data);

/*
 * Funcția generează un puls pe semnalul R_CLOCK (row clock), care trece
 * la urmatorul rând al matricei.
 */
void pulse_row_clock(void);

/*
 * Funcția activează semnalul de master reset, folosit pentru a reseta
 * complet registrul de deplasare / shiftare.
 */
void master_rst(void);

/*
 * Funcția transmite datele corespondente LED-urilor care trebuie aprinse
 * pe matricea de LED-uri, utilizând interfața SPI, cu câte un puls pe
 * fiecare linie pentru actualizarea datelor.
 * Parametrul delay_cycles controlează viteza de refresh, permițând o
 * afișare cu ritm variabil. Parametrii left și right indică indicii
 * pattern-urilor pentru afișajul stâng și drept.
 */
void transmitere_afisare(int64_t delay_cycles, uint8_t left, uint8_t
right);

#endif
```

`matrix.c`

Fișierul **`matrix.c`** definește toate funcțiile de control al matricei **LED**, inclusiv inițializarea pinilor și **SPI-ul** (**`matrix_init()`**), logica de trimitere a valorilor binare pentru fiecare linie a cifrelor (folosind matricea **`digits[10][7]`** pentru crearea simbolurilor corespunzătoare cifrele 0-9), trimiterea acestor date către registrele de deplasare prin **SPI** și generarea semnalelor necesare pentru comutarea rândurilor, afișând astfel 2 cifre simultan pe matrice.

```
/*----------------------------------------------------------------------
 * Fișier: matrix.c
 * Utilizat pentru definirea funcțiilor și a variabilelor folosite pentru
 * matricea de LED-uri
 *--------------------------------------------------------------------*/

/*----------------------------------------------------------------------
 * Includes
 *--------------------------------------------------------------------*/

// General
#include "matrix.h"
#include "spi.h"
```

```
/*---------------------------------------------------------------------
 * Public defines
 *--------------------------------------------------------------------*/

// Set/Unset latch
#define SET_LATCH() (PORTK |= (1 << PK4))
#define UNSET_LATCH() (PORTK &= ~(1 << PK4))

// Set/Unset r_rst
#define SET_PULSE_ROW_RST() (PORTL |= (1 << PL3))
#define UNSET_PULSE_ROW_RST() (PORTL &= ~(1 << PL3))

// Set/Unset r_clk
#define SET_PULSE_ROW_CLK() (PORTF |= (1 << PF2))
#define UNSET_PULSE_ROW_CLK() (PORTF &= ~(1 << PF2))

// Set/Unset master_rsts
#define SET_MASTER_RST() (PORTK |= (1 << PK5))
#define UNSET_MASTER_RST() (PORTK &= ~(1 << PK5))

/*---------------------------------------------------------------------
 * Public variables
 *--------------------------------------------------------------------*/

// Matrice cu reprezentarea cifrelor de la 0 la 9 (7 linii / cifră)
uint8_t digits[10][7] = {
  {0x70, 0x88, 0x88, 0x88, 0x88, 0x88, 0x70}, // 0
  {0x08, 0x18, 0x28, 0x48, 0x08, 0x08, 0x08}, // 1
  {0x70, 0x88, 0x08, 0x10, 0x20, 0x40, 0xF8}, // 2
  {0x70, 0x88, 0x08, 0x70, 0x08, 0x88, 0x70}, // 3
  {0x10, 0x30, 0x50, 0xF8, 0x10, 0x10, 0x10}, // 4
  {0xF8, 0x80, 0x80, 0x70, 0x08, 0x88, 0x70}, // 5
  {0x70, 0x88, 0x80, 0xF0, 0x88, 0x88, 0x70}, // 6
  {0xF8, 0x08, 0x10, 0x20, 0x20, 0x20, 0x20}, // 7
  {0x70, 0x88, 0x88, 0x70, 0x88, 0x88, 0x70}, // 8
  {0x70, 0x88, 0x88, 0x78, 0x08, 0x08, 0x70}  // 9
};

/*---------------------------------------------------------------------
 * Private functions
 *--------------------------------------------------------------------*/

/*
 * Funcția gpio_init:
 * - Configurează pinii care controleaza latch-ul, r_clock și r_rst
 * - Inițializează latch-ul pe HIGH
 */
private void gpio_init(void) {
  DDRK |= (1 << PK4);  // PK4 - LATCH (output)
  DDRF |= (1 << PF2);  // PF2 - R_CLOCK (output)
  DDRL |= (1 << PL3);  // PL3 - R_RST (output)
  PORTK |= (1 << PK4); // Inițializare latch pe HIGH
}

/*---------------------------------------------------------------------
 * Public functions
 *--------------------------------------------------------------------*/
```

```
/*
 * Funcția matrix_init:
 * - Inițializează pinii de control pentru matricea de LED-uri
 * - Inițializează interfața SPI necesară transmiterii datelor
 */
void matrix_init(void) {
  gpio_init();
  spi_init();
}

/*
 * Funcția pulse_latch:
 * - Generează un puls LOW-HIGH pe pinul latch pentru a trimite datele la
 * ieșire
 */
void pulse_latch(void) {
  UNSET_LATCH();
  __delay_cycles(100);
  SET_LATCH();
}

/*
 * Funcția pulse_row_rst:
 * - Generează un puls pe R_RST pentru a reseta selecția la primul rând
 */
void pulse_row_rst(void) {
  SET_PULSE_ROW_RST();
  __delay_cycles(100);
  UNSET_PULSE_ROW_RST();
}

/*
 * Funcția pulse_row_clock:
 * - Generează un puls pe R_CLOCK pentru a avansa la următorul rând al
 *   matricei
 */
void pulse_row_clock(void) {
  SET_PULSE_ROW_CLK();
  __delay_cycles(100);
  UNSET_PULSE_ROW_CLK();
}
/*
 * Funcția master_rst:
 * - Generează un puls de reset pe un pin dedicat master reset (dacă
 *  acesta este conectat)
 */
void master_rst(void) {
  SET_MASTER_RST();
  __delay_cycles(100);
  UNSET_MASTER_RST();
}

/*
 * Funcția transmite pe matricea de LED-uri datele apartenente formelor
 * grafice (ex: săgeți) specificate de parametrii left și right,
utilizând
 * interfața SPI.
 * Afișarea se face linie cu linie, controlând manual poziția rândului
 * curent prin semnale de ceas, cu un delay variabil între actualizări,
 * determinat de delay_cycles.
 */
```

```
void transmitere_afisare(int64_t delay_cycles, uint8_t left, uint8_t
right) {
  // Parcurge fiecare dintre cele 7 linii ale matricii
  for (int i = 0; i < 7; i++) {
    // Selectează linia curentă: resetează poziția și avansează cu i pași
    pulse_row_rst(); // Revine la începutul rândurilor
    for (int j = 0; j < i; j++) {
      pulse_row_clock(); // Avansează cu un rând
    }
    /*
     * Transmite câte un byte pentru fiecare jumătate a matricii:
     * - left: pattern-ul pentru jumătatea stângă (ex: săgeata stângă)
     * - right: pattern-ul pentru jumătatea dreaptă (ex: săgeata dreaptă)
     */
    spi_transmit(digits[left][i]);   // Coloane pentru partea stângă
    spi_transmit(digits[right][i]);  // Coloane pentru partea dreaptă
    // Actualizează ieșirea LED-urilor după ce datele au fost trimise
    pulse_latch();
    /*
     * Introduce o întârziere variabilă între afișările liniilor,
     * controlată din funcția main() prin parametrul delay_cycles.
     */
    volatile int32_t k = 0;
    for (k = 0; k < delay_cycles; ++k) {
      __no_operation(); // Consumă timp (1 ciclu de ceas per
instrucțiune)
    }
  }
}
```

9.10.1 AFIȘAREA SIMBOLURILOR

Cerință: Să se realizeze o aplicație care aprinde progresiv **LED-urile** unei **matrice 7x10**, într-o secvență accelerată, astfel încât acestea să formeze în final **2 săgeți** orientate spre exterior. Comunicarea cu **matricea de LED-uri** se va realiza prin interfața **SPI** configurată în modul **Master**, trimițând datele **în mod blocant** (fără întreruperi), linie cu linie. Pe măsură ce secvența evoluează, timpul de așteptare dintre actualizările succesive ale liniilor **va scădea**, generând un efect vizual de accelerare.

Sugestii: *Pentru implementare se vor folosi două registre de shiftare conectate în paralel, fiecare corespunzând unei jumătăți a matricei (stânga și dreapta). Fiecare registru va controla o coloană specifică pentru un anumit rând al matricii. Un contor intern va selecta rândul curent, iar datele vor fi transmise sincron, în funcție de modelul grafic al săgeților. Temporizarea între pașii de afișare se va reduce treptat pentru a obține efectul de „accelerare" a aprinderii LED-urilor.*

Conectări hardware: *Conectările se fac conform indicațiilor din laborator, iar în cazul codului de mai jos,* ***matricea de LED-uri*** *se conectează pe* ***microBUS 2****. Matricea se poate conecta,* ***la voia studentului****, pe orice alt microBUS,* ***cu condiția*** *modificării pinilor aferenți microBUS-ului respectiv.*

main.c

main.c este punctul de intrare al aplicației, unde se inițializează **matricea de LED-uri** și interfața **SPI**. La pornire, se rulează o animație cu aprinderea progresiv accelerată a **LED-urilor**, care formează **2 săgeți** orientate spre exterior (◀ ▶). Viteza de afișare crește treptat prin reducerea controlată a unei întârzieri (**current_delay**). Afișarea se face blocant, linie cu linie, prin funcția **transmitere_afisare()**.

```
/*------------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul principial de rulare a aplicației cu săgeți
 *----------------------------------------------------------------------*/

#include "matrix.h"
#include "spi.h"

/*
 * Program principal:
 * Afișeaza 2 săgeți în oglindă într-un ritm tot mai accelerat pentru a
 * da iluzia unei afișări complete.
 */

/*------------------------------------------------------------------------
 * Global variables
 *----------------------------------------------------------------------*/

int64_t current_delay = 0;
void main(void) {

  // Inițializare hardware
  matrix_init();
  __delay_cycles(100); // Scurtă pauză pentru stabilitate

  while(1){

    // Viteza minimă / delay-ul maxim la care va ajunge afișarea (cu cât
    // viteza e mai mare, cu atât afișarea e mai lentă)
    int32_t max_arrow_delay = 30000; //1875 µs = 1,875 ms

    // Viteza maximă / delay-ul minim la care va ajunge afișarea (cu cât
    // viteza e mai mică, cu atât afișarea e mai rapidă)
    const int32_t min_arrow_delay = 500; //0.03125 ms = 31.25 µs
    int procent = 20;
    current_delay = max_arrow_delay;

    // Loop cu 200 de iterații în care scădem delay-ul la fiecare pas.
    for (int i = 0; i < 100 && current_delay > 500; i++) {
      current_delay -= (procent * current_delay ) / 100;
      if(current_delay <= min_arrow_delay * 10)
        procent = 2;

    // Afișează săgețile la viteza curentă
      transmitere_afisare(current_delay, 11, 10);
    }

    for (int i = 0; i < 50; i++)
      transmitere_afisare(current_delay, 11, 10);
  }
}
```

9.10.2 COUNTER DE LA 0 LA 99 (BLOCANT / FĂRĂ ÎNTRERUPERI)

Cerință: Să se realizeze o aplicație care afișează un **contor cu valori de la 0 la 99** pe o **matrice de LED-uri 7x10**, utilizând **interfața SPI** în modul **Master** pentru a transmite datele. Se vor afișa pe fiecare jumătate de matrice cifra zecilor, respectiv a unităților corespunzătoare contorului. Transmiterea datelor către matrice se face **în mod blocant** (fără întreruperi), linie cu linie, actualizând secvențial afișajul în timp real.

Sugestii: *Pentru afișare trebuie folosite cele 2 registre de shiftare, luând în considerare faptul că una se ocupă de rândul de pe prima jumătate de matrice, iar cealaltă de rândul de pe cealaltă jumătate, counter-ul ocupându-se de schimbarea rândurilor.*

Conectări hardware: *Conectările se fac conform indicațiilor din laborator, iar în cazul codului de mai jos, **matricea de LED-uri** se conectează pe **microBUS 2**. Matricea se poate conecta, **la voia studentului**, pe orice alt microBUS, **cu condiția** modificării pinilor aferenți microBUS-ului respectiv.*

`main.c`

`main.c` este punctul de intrare al aplicației, unde se inițializează matricea și interfața SPI, apoi într-o buclă infinită se apelează periodic funcția **`transmitere_afisare()`** pentru a afișa valorile contorului de la 0 la 99, controlând ritmul de creștere cu ajutorul unui contor auxiliar pentru întârziere.

```
/*----------------------------------------------------------------------
 * Fișier: main.c
 * Fișierul principial de rulare a aplicației counter
 *--------------------------------------------------------------------*/

#include "matrix.h"
#include "spi.h"

/*
 * Program principal:
 * Afișează 2 săgeți în oglindă într-un ritm tot mai accelerat pentru a
 * da iluzia unei afișări complete.
 */

/*----------------------------------------------------------------------
 * Global variables
 *--------------------------------------------------------------------*/

int64_t current_delay = 0;


void main(void) {
  // Inițializare hardware
  matrix_init();
  __delay_cycles(100); // Scurtă pauză pentru stabilitate

  // Contor cu viteză constantă
  uint8_t counter = 0; // Valoarea contorului (00-99)
  uint8_t j = 0;       // Contor intermediar pentru a controla viteza de
                       // schimbare a numerelor

  while (1) {
    uint8_t u = 0, z = 0; // Cifrele unităților și zecilor

    // Descompune contorul în cifrele individuale
    u = counter % 10;  // Cifra unităților
    z = counter / 10;  // Cifra zecilor
    // Afișeaza numărul curent.
    transmitere_afisare(500, u, z);
```

```
    // Bucla 'j' controlează cât de repede se schimbă cifrele
    j++;
    if (j == 50) { // Schimbă nr. la fiecare 50 de refresh-uri complete
      j = 0;
      counter++;
    }

    // Resetează contorul după ce ajunge la 99
    if (counter > 99) {
      counter = 0;
    }
  }
}
```

9.11 BIBLIOGRAFIE

1. *"8-bit AVR Microcontroller ATmega 1280 Datasheet", Microchip Technology*
2. *"Schematic for UNI Clicker", Mikroe*
3. *"Schematic for ATmega1280: SiBrain", Mikroe*
4. *"Serial Peripheral Interface", Wikipedia*
5. *"Introduction to SPI Interface", Analog Devices*
6. *"SPI Basics", Circuit Basics*

10. INTER-INTEGRATED CIRCUITS – I²C

10.1 UNDE SE FOLOSEȘTE I²C?

I²C este utilizat pe scară largă în dispozitive electronice moderne, fiind ideal pentru comunicarea între mai multe componente folosind doar două fire. Printre cele mai comune aplicații se numără:

- **Senzori:** transmiterea valorilor măsurate către microcontroller;
- **Memorii EEPROM:** stocarea și citirea datelor nevolatile în sisteme embedded;
- **Ceasuri în timp real (RTC):** sincronizarea precisă a timpului cu microcontroller-ul;
- **Display-uri OLED / LCD mici:** controlul afișajului pentru ceasuri și dispozitive portabile;
- **Convertori ADC/DAC:** schimb de date analog-digitale cu microcontrolerul prin magistrala I²C.

Figura 10.1 – Un senzor I²C de presiune și umiditate (stânga) și un display I²C LCD (dreapta)

10.2 CUNOȘTINȚE NECESARE

Pentru a putea parcurge acest capitol, este necesar să cunoașteți următoarele subiecte din capitolele anterioare, și nu numai:

- Utilizarea întreruperilor;
- Utilizarea timerelor pentru întreruperi sincronizate și generarea semnalelor PWM;
- Modul de funcționare al memoriilor EEPROM;
- Utilizarea kit-ului hardware Mikroe.

10.3 ABSTRACT

Acest capitol detaliază protocolul de comunicație **I²C** (**Inter-Integrated Circuit**) și implementarea sa prin interfața hardware **TWI** (**Two-Wire Interface**) disponibilă în microcontroller-ul **ATmega1280**. Pentru a exemplifica modul de funcționare al protocolului și al blocului hardware, s-a propus implementarea unei aplicații folosind memoria **EEPROM click**, un **USB** to **I²C click** și un **16x9G click**.

10.4 HARDWARE / SOFTWARE NECESAR

- ATmega1280;
- UNI clicker;
- Atmel ICE;
- EEPROM click;
- USB to I²C click;
- 16x9 G click;
- Osciloscop;
- IAR Embedded Workbench 7.30.5.

10.5 EEPROM CLICK

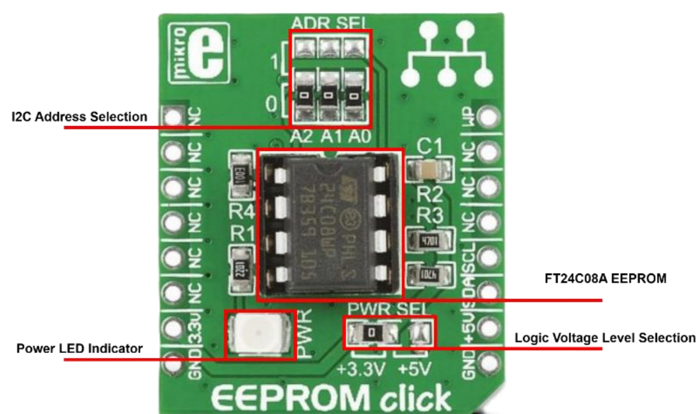


Figura 10.2 – EEPROM Click

EEPROM Click utilizează circuitul **FT24C08A**, o memorie **EEPROM** serială de **8 Kb** (**8x1024 biți**), organizată în **64 pagini** a câte **16 octeți** fiecare. Comunicarea se face prin interfața **I²C** (pinii **SDA** și **SCL**).

Pe placa **UNI Clicker** cu **ATmega1280**, **EEPROM** se conectează printr-un socket **mikroBUS**, unde:
- **SDA** (**I²C Data**) de pe **EEPROM Click** se conectează la pinul **PD1** (**SDA**) al **ATmega1280**.
- **SCL** (**I²C Clock**) de pe **EEPROM Click** se conectează la pinul **PD0** (**SCL**) al **ATmega1280**.

Astfel, accesarea memoriei externe se face prin modulul hardware **TWI** (**Two-Wire Interface**) integrat în **ATmega1280**.

10.5.1 FUNCȚIONALITĂȚI PRINCIPALE

- Capacitate **8 Kb**;
- Organizare internă în pagini de **16 octeți**;
- Protecție la scriere prin pinul **WP** (conectat la **PWM** pe **mikroBUS**, poate fi controlat și din **ATmega** dacă e necesar);
- Alimentare compatibilă cu **3.3 V** și **5 V** (selectabilă prin **jumper** pe placă);
- Adresă **I²C** configurabilă prin jumperi **A0–A2** (default **0x50–0x57**, în funcție de setare).

10.5.2 INTEGRARE SOFTWARE PE ATMEGA1280

ATmega1280 are implementat **TWI hardware**, compatibil **100/400 kHz** (**Standard** / **Fast Mode**), deci **EEPROM Click** suportă până la **1 MHz** (**Fast Plus**). Comunicarea se face prin transmiterea adresei **Slave** (**0x50 + A2..A0**) urmată de adresa internă pe **10 biți** și apoi datele efective.

Se pot folosi funcții dedicate pentru:
- scriere pagină (**16 octeți** maxim per tranzacție);
- citire secvențială (mai multe locații la rând);
- activare protecție scriere (**WP = 1**).

10.5.3 EXEMPLE DE UTILIZARE CU ATMEGA1280

1. **Inițializare TWI (I²C)** la **100 kHz** sau **400 kHz**;
2. **Scriere:** se transmite adresa memoriei, urmată de **1–16 octeți de date**;
3. **Citire:** se transmite adresa internă de start, apoi se citește secvențial un număr de octeți;
4. **Protecție scriere:** se setează pinul **WP HIGH** prin pinul corespunzător de pe **mikroBUS**, dacă se dorește protecția memoriei.

10.6 USB TO I^{2}C CLICK

Figura 10.3 – USB to I^{2}C Click

USB to I^{2}C Click are la bază circuitul **FT260Q** de la **FTDI**, un dispozitiv **USB 2.0 Full-Speed** care permite conversia **USB-I^{2}C** și **USB-UART**, folosind clasele **HID** standard (**Human Interface Device**). Avantajul major este că aceste clase sunt recunoscute nativ de majoritatea sistemelor de operare (**Windows**, **Linux**, **macOS**) și nu necesită drivere speciale.

10.6.1 FUNCȚIONALITĂȚI PRINCIPALE

- Bridge **USB-I^{2}C** și **USB-UART**, cu ambele interfețe disponibile simultan;
- **I^{2}C Master** cu viteze de până la **3.4 Mbit/s**;
- **UART** cu **RX**, **TX** și control hardware (**CTS**, **RTS**);
- **USB 2.0 Full-Speed** (**12 Mbps**), protocol complet implementat în hardware;
- Reset hardware (**pin RST**) și funcție de remote wake-up (**pin INT**);
- Alimentare selectabilă **3.3 V** sau **5 V**, compatibilă cu logica **ATmega1280**;
- Protecție la **back-voltage** pe liniile de alimentare (**MAX40200**);
- **LED-uri de stare**: **PWR** (alimentare activă) și **USB IN** (alimentare USB activă).

10.6.2 INTEGRARE SOFTWARE PE ATMEGA1280

USB to I^{2}C Click poate fi folosit în două moduri distincte:

1. **Bridge USB-I^{2}C**
   - **FT260Q** funcționează ca **Master I^{2}C**, controlând senzori sau memorii conectate pe **I^{2}C**.
   - **ATmega1280** poate fi configurat ca **Slave I^{2}C**, permițând schimb de date direct prin **USB**.

2. **Bridge USB-UART**
   - Conectarea pinilor **TX, RX, CTS, RTS** la unul dintre **USART-urile ATmega1280** permite comunicația **directă** cu PC-ul **prin USB**.
   - Util pentru **debugging** și trimiterea de mesaje seriale către PC fără convertor **UART-USB** extern.

10.6.3 FUNCȚII SUPLIMENTARE

- **Wake-up prin USB:** dacă PC-ul suportă remote wake-up, placa poate trezi gazda prin pinul **INT**.
- **Reset hardware:** prin pinul **RST**, **FT260Q** poate fi resetat extern.
- **Standalone mode:** placa poate fi alimentată doar USB, funcționând independent de microcontroler.

10.6.4 BENEFICII

- **Plug-and-play** pe orice sistem de operare datorită clasei **HID** standard;
- Posibilitate de **testare rapidă a senzorilor I^{2}C** fără a scrie cod pe microcontroler;
- **Flexibilitate maximă:** poate lucra fie cu **ATmega1280**, fie direct cu **PC-ul**;
- **Viteză mare pe I^{2}C (3.4 MHz)**, potrivită pentru periferice rapide.

10.7 16X9 G CLICK

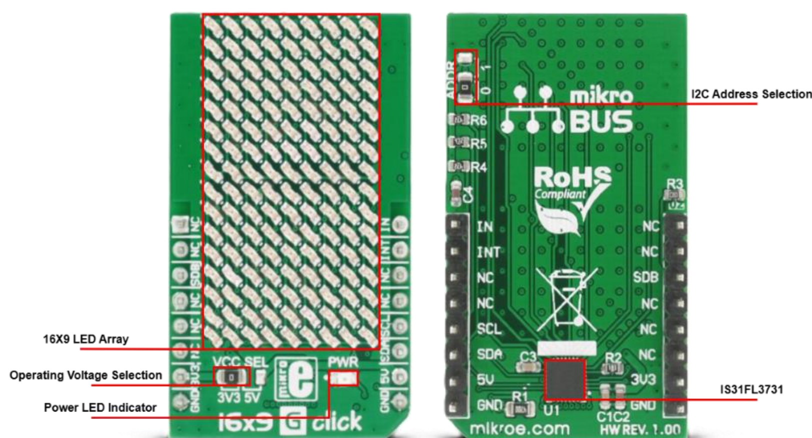


Figura 10.4 – 16x9 G Click

16x9 G Click este o placă de extensie bazată pe matricea LED de 16 coloane × 9 rânduri (144 LED-uri verzi) controlată de driver-ul **IS31FL3731**. Aceasta permite control individual pentru fiecare LED (pornire/oprire și intensitate luminoasă), stocarea mai multor cadre (frames) și rularea de animații autonome sau sincronizate cu un semnal audio.

10.7.1 FUNCȚIONALITĂȚI PRINCIPALE

- **Matrice 16x9** cu **144 LED-uri** verzi;
- **8 cadre (frames) memorabile** pentru animații;
- Control **individual** al fiecărui LED;
- Reglare fină a intensității (**PWM per pixel**);
- Control **prin I^2C** (**max. 400 kHz**);
- Alimentare: **3.3 V sau 5 V**;
- Compatibil cu standardul **mikroBUS™**;
- **Driver IS31FL3731** cu 3 moduri de lucru:
  1. **Picture Mode** – se afișează un singur cadru din memoria internă;
  2. **Auto Frame Play Mode** – derulare automată a până la 8 cadre, pentru animații simple;
  3. **Audio Frame Play Mode** – animațiile sunt modulate după intensitatea unui semnal audio.

10.7.2 INTEGRARE CU ATMEGA1280

1. Se pornește interfața **I^2C** la **100 kHz** sau **400 kHz**;
2. Se trimite comanda pentru selectarea cadrului curent (**FRAME register** = **0x00…0x07**);
3. Se scriu datele pentru fiecare **coloană / rând** în buffer-ul intern al driverului;
4. Se apelează **update display** pentru a activa conținutul cadrului;
5. Se configurează **Auto Play Mode** sau **Audio Play Mode** pentru animații (opțional).

10.7.3 EXEMPLE DE UTILIZARE

- **Afișare statică:** aprinderea unui **LED** la coordonatele (**x = 0**, **y = 0**);
- **Blink LED:** activarea modului **blink** pentru o **coloană** sau un **pixel**;
- **Animații:** scrierea mai multor cadre și rularea lor automat cu **Auto Play Mode**;
- **Spectru audio vizual:** conectarea intrării audio și folosirea **Audio Frame Play Mode** pentru efecte sincronizate cu muzica.

10.7.4 BENEFICII

- Permite implementarea rapidă a **display-urilor vizuale simple** în aplicații embedded (aparate electrocasnice, IoT, gadgeturi portabile);
- Posibilitate de **animații complexe** fără suprasolicitarea **ATmega1280**;

- Control fin al **intensității** și **blink-ului** pentru **fiecare LED** individual;
- Suport nativ pentru **efecte audio**, unic față de alte matrici LED clasice.

10.8 FUNCȚIONALITATE

ATmega1280 poate gestiona simultan **EEPROM Click**, **USB to I^{2}C Click** și **16x9 G Click** pentru a implementa aplicații complexe de stocare și afișare vizuală. Fiecare **click** aduce funcționalități complementare care pot fi orchestrate prin magistrala **I^{2}C** și interfețele digitale ale microcontrolerului.

10.8.1 CONEXIUNI ȘI ROLURI

- **EEPROM Click**: memorie non-volatilă, folosită pentru **salvarea datelor**, **configurărilor** sau **stărilor LED-urilor**. Comunicarea se face prin **I^{2}C** (**SDA / SCL**);
- **16x9 G Click**: matrice **LED**, care afișează vizual **datele** sau **stările stocate** în **EEPROM**. Controlul fiecărui **LED** și al **cadrelor** se face tot prin **I^{2}C**;
- **USB to I^{2}C Click**: poate conecta **ATmega1280** la un **PC** sau alt dispozitiv **USB**, permițând transmiterea **datelor** către / de la **EEPROM** și **matricea LED**. Acesta poate fi **Master I^{2}C** sau interfață **UART** pentru **debug**.

Astfel, microcontrolerul poate citi / seta date în **EEPROM**, trimite aceste date ca **pattern** pentru **matricea LED** și recepționa comenzi de la **PC** prin **USB to I^{2}C Click** pentru **actualizarea** în timp real a **display-ului**.

10.8.2 FLUX DE DATE TIPIC

1. Stocare inițială: **ATmega1280** scrie **configurații** sau **modele de LED-uri** în **EEPROM Click**;
2. Citire și afișare: **ATmega1280** citește **datele** din **EEPROM** și le trimite la **16x9 G Click** pentru a aprinde **LED-urile** corespunzătoare.
3. Interacțiune cu PC: **USB to I^{2}C Click** primește comenzi prin **USB**, cum ar fi **"aprinde LED-ul (x = 2, y = 3)"**, iar **ATmega1280** procesează comanda și **actualizează matricea**.
4. Actualizare automată: se pot folosi modurile **Auto Frame Play** sau **Audio Frame Play** pentru animații sau efecte vizuale **sincronizate** cu **semnalul audio**, folosind date stocate anterior în **EEPROM** sau primite prin **USB**.

10.9 DIFERENȚELE ÎNTRE ATMEGA1280 ȘI ATMEGA3250

În ceea ce privește integrarea și suportul pentru diferite funcționalități hardware, microcontrolerele pot prezenta numeroase diferențe de la un model la altul. Un exemplu relevant în contextul de față este chiar diferența dintre **ATmega1280** și **ATmega3250** cu privire la modul de implementare al comunicării **I^{2}C**.

ATmega1280 include un modul **TWI** (**Two-Wire Interface**) hardware **dedicat**, complet automatizat, care gestionează **direct** protocolul **I^{2}C**. În schimb, **ATmega3250 nu are** un **modul TWI** propriu-zis, ci integrează un bloc **USI** (**Universal Serial Interface**), care poate fi configurat prin **software** pentru a implementa comunicația **I^{2}C**, cu un grad mai mare de **flexibilitate**, dar și **complexitate în cod**. De asemenea, **ATmega1280 nu dispune** de **modul USI**.

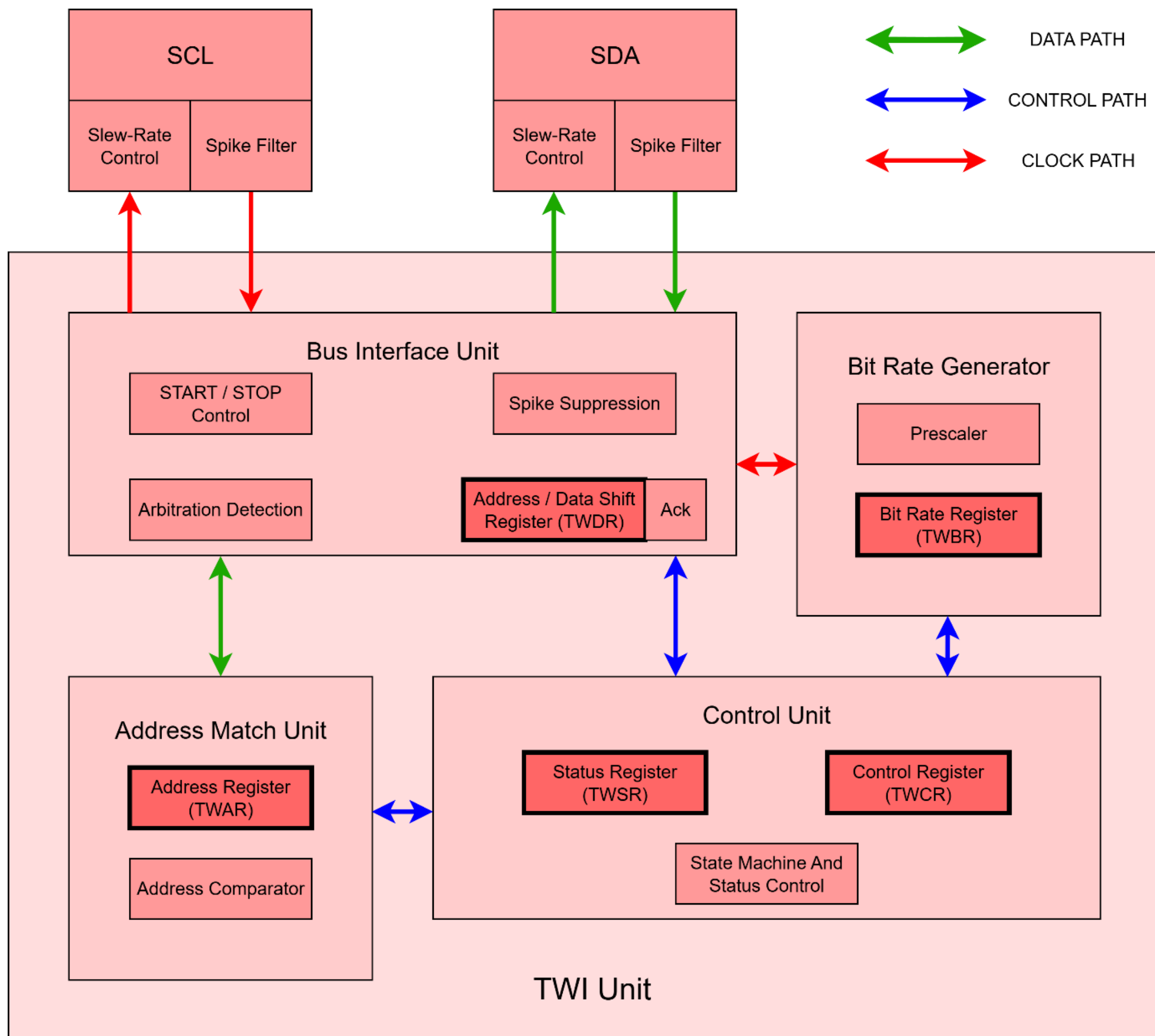


Figura 10.5 – Modulul TWI prezent în ATmega1280

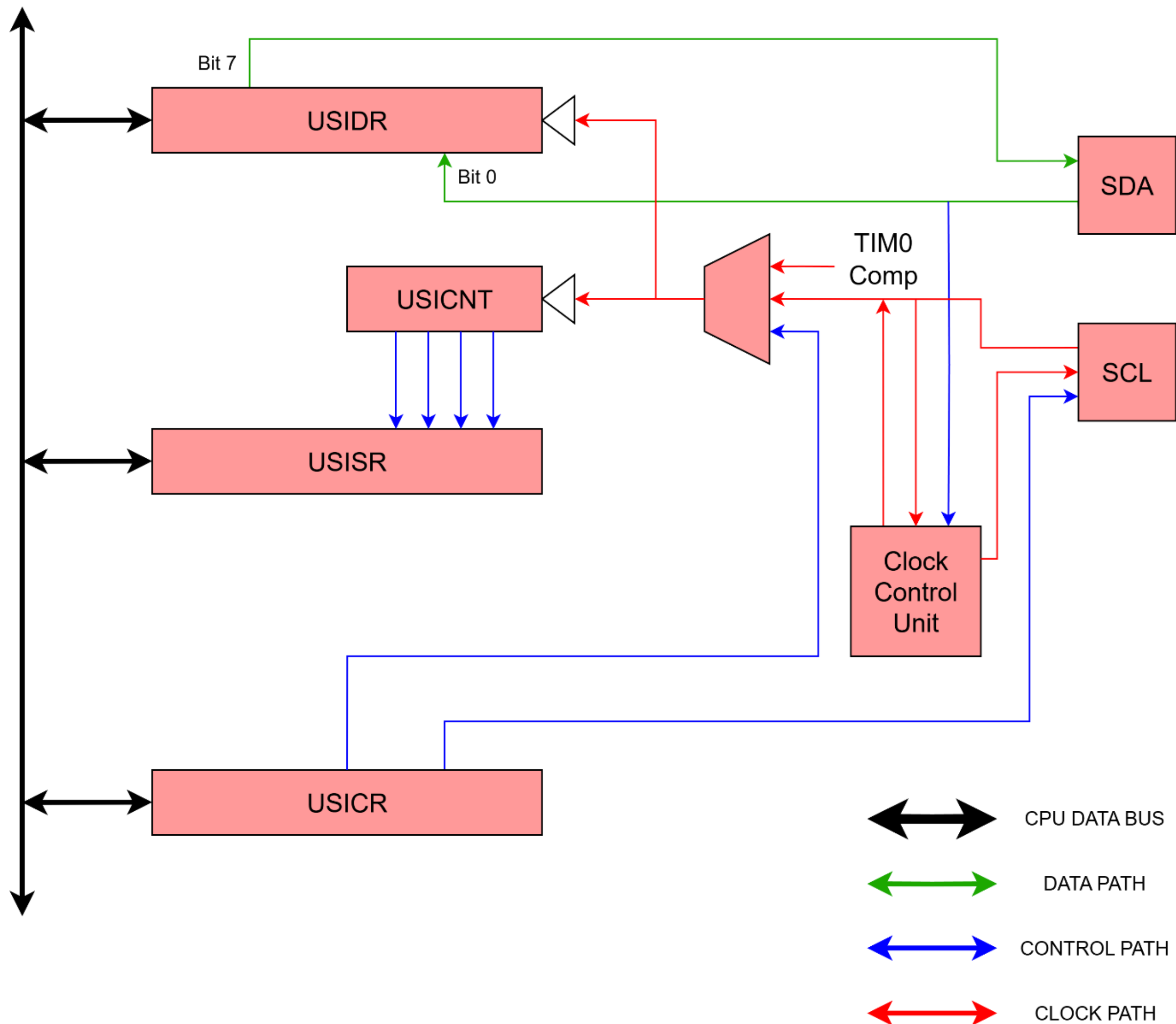


Figura 10.6 – Modulul USI prezent în ATmega3250

10.10 INTRODUCERE ÎN I²C

Definiție

I²C (Inter-Integrated Circuit) este un protocol de transmisie serial de tip Master-Slave, folosit pentru a permite comunicarea între dispozitivele electronice integrate.

Acest tip de transmisie a fost inventat în anul **1982** de către divizia de circuite semiconductoare **NXP** a companiei olandeze **Philips**. Pe parcursul dezvoltării circuitelor integrate, protocolul **I²C** a suferit mai multe schimbări regăsite în următoarele versiuni:

- **1982**: **Versiunea inițială** a sistemului **I²C** folosit pentru transmisia de informații între diversele circuite implementat de Philips;
- **1992**: **Versiunea 1.0** – prima versiune standardizată care a adăugat, pe lângă modul standard de **100 kHz** şi aşa-numitul **Fast Mode (Fm) de 400 khz**. De asemenea a fost modificat şi modul de adresare, acesta fiind trecut pe **10 biţi** crescând astfel capacitatea nodurilor suportate de protocol;
- **1998**: **Versiunea 2.0** – a adăugat modul **High Speed 3.4 MHz**;
- **2000**: **Versiunea 2.1** – implementează mici modificări de mentenanţă a versiunii anterioare;
- **2007**: **Versiunea 3.0** – a adăugat modul **Fast-Mode Plus** (**Fm+**);
- **2013**: **Versiunea 4.0** – a adăugat modul **Ultra Fast-Mode** (**UFm**) pentru noile canale **USDA** si **USCL** care foloseau logica de tip **push-pull** fără a mai fi nevoie de rezistenţe de pull-up.

I²C foloseşte ca mediu de transmisie numai **2 linii de bus bidirecţionale**, una pentru **pachetele de date** (**SDA**) şi una pentru **clock** (**SCL**). De asemenea, pentru fiecare linie a bus-ului **I²C** este nevoie de o **singură rezistenţă** de pull-up conectată la sursa de alimentare. Tensiunile tipice folosite au valorile de **+5 V** sau de **+3.3 V**, deşi sunt permise şi alte valori.

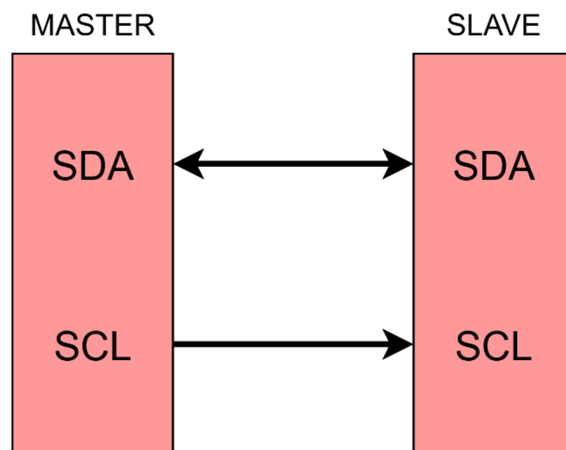


Figura 10.7 – Comunicarea I²C între Master şi Slave

Un dispozitiv conectat într-o reţea **I²C** poate avea două roluri: cel de **Master** şi respectiv, cel de **Slave**, acest dispozitiv purtând denumirea de nod. Astfel, un nod **Master** are rolul de a iniţia comunicarea cu nodurile **Slave** şi de a genera clock-ul. Un nod **Slave** recepţionează clock-ul de la un dispozitiv de tip **Master** şi răspunde la cererile acestuia.

Pentru un nod dintr-o conexiune **I²C** există patru moduri de operaţie posibile:

1. Master Transmitter – nodul **Master** transmite mesaje către un nod **Slave**;
2. Master Receiver – nodul **Master** recepţionează date de la un nod **Slave**;
3. Slave Transmitter – nodul **Slave** transmite mesaje către un nod **Master**;
4. Slave Receiver – nodul **Slave** recepţionează date de la un nod **Master**.

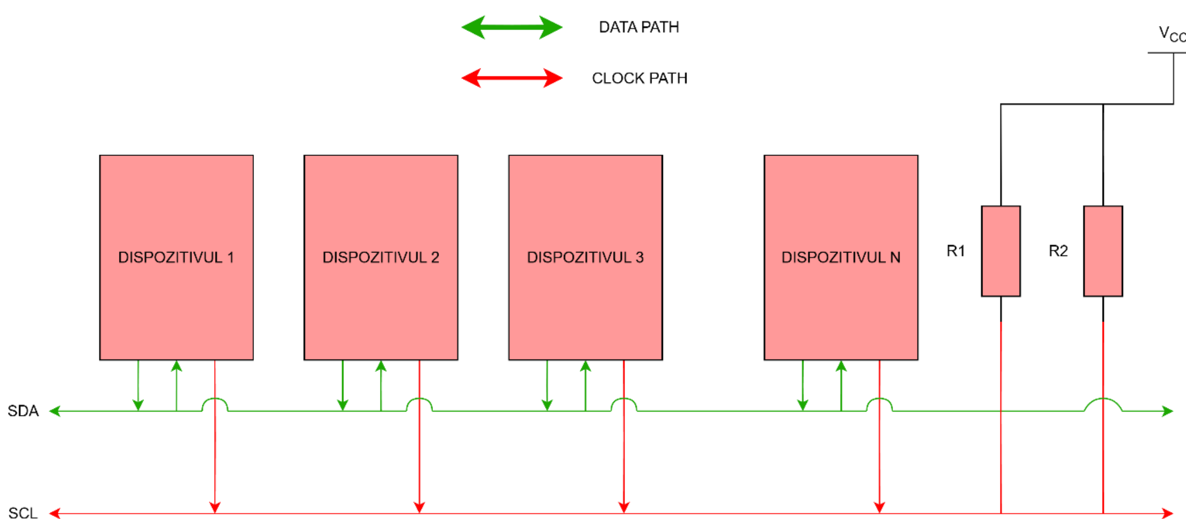


Figura 10.8 – Conectarea dispozitivelor în protocolul I²C

10.11 INTERFAŢA TWI (TWO WIRE INTERFACE)

Acest tip de interfaţă este compatibilă cu protocolul **I²C**, având un mod de adresare pe **7 biţi**, permiţând utilizatorului să conecteze într-o reţea **I²C** până la **128 de dispozitive** pe acelaşi bus de date. Nodurile din această reţea sunt capabile de a transmite date atât în modul **standard** (**<100 kBps**) cât şi în modul **Fast** (**<400 kBps**).

10.11.1 TRANSMISIA DE DATE FOLOSIND INTERFAŢA I²C

O transmisie de tip **TWI** constă dintr-un bloc de start, un bloc indicator de **Read / Write**, confirmarea de la dizpozitivul **Slave**, unul sau mai multe pachete de date şi un bloc de stop. Fiecare bit din mesajele vehiculate pe bus-ul **TWI** este însoţit de un puls pe linia de clock. Pentru a asigura validitatea datelor, tensiunea liniei de transmisie a datei trebuie sa rămână stabil pe nivelul logic **HIGH**, excepţie făcând cazurile când sunt generate condiţiile de **START/STOP**.

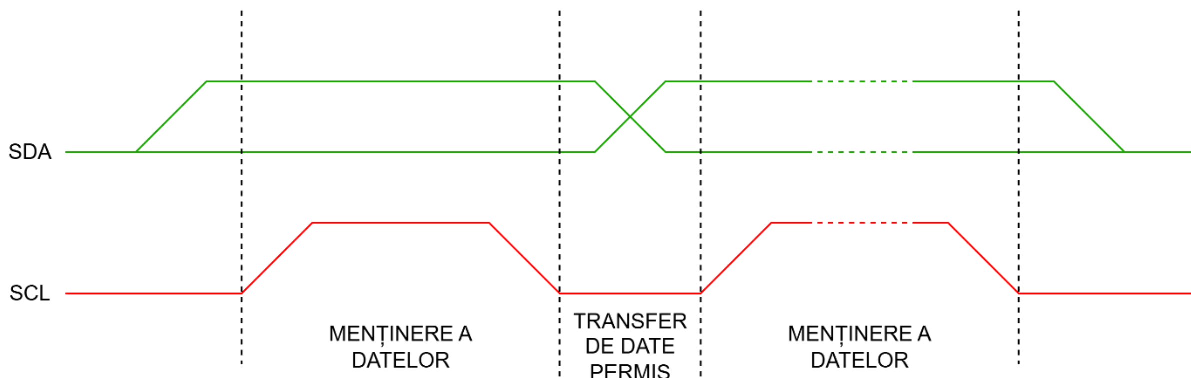


Figura 10.9 – Validitatea datelor la transmisia TWI

10.11.2 CONDIŢIILE DE START / STOP

Unul dintre rolurile unui nod **Master** este de a iniţia şi de a încheia o transmisie pe bus-ul TWI. Transmisia este iniţiată când dispozitivul **Master** emite o condiţie de **START**, şi este încheiată atunci când este emisă condiţia de **STOP**. Între aceste două condiţii bus-ul este considerat ocupat, deci nici un alt **Master** nu poate accesa bus-ul în acest interval de timp. Singura excepţie în acest caz este atunci când între emisia condiţiilor de **START/STOP** se emite o noua condiţie de **START**. Această condiţie se numeşte **REPEATED START** şi este folosită de un nod de tip **Master** care doreşte sa reiniţieze un transfer fără a elibera controlul bus-ului TWI. După emisia unei condiţii de tip **REPEATED START** bus-ul este ocupat până la apariţia condiţiei de **STOP**. Condiţiile de **START/STOP** sunt evidenţiate prin schimbarea nivelului logic a liniei SDA atunci când SCL este pe nivelul logic **HIGH**, cum se poate observa în **Figura 10.10**.

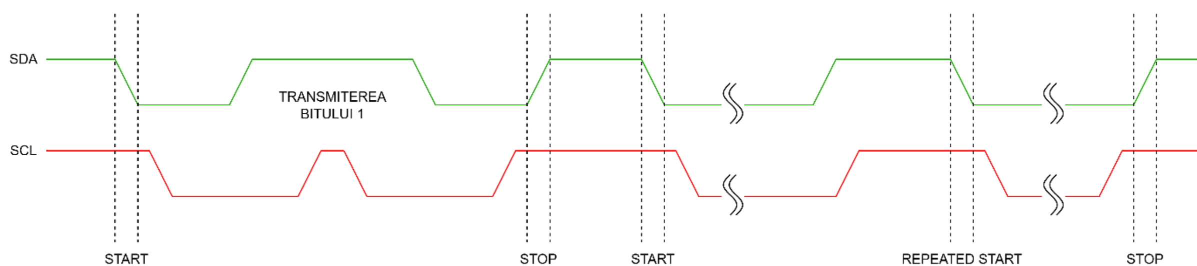


Figura 10.10 – Condiţiile de START, STOP şi REPEATED START

10.11.3 FORMATUL PACHETELOR DE ADRESE

Toate pachetele de adresă vehiculate pe bus-ul **TWI** au lungimea de **9 biţi**, dintre care **7 biţi** sunt cei de adresă, un bit de **control al operaţiei** de **READ/WRITE** şi un bit de **confirmare**. Dacă se va seta bitul de **READ/WRITE** pe **HIGH**, atunci se va executa o operaţie de citire, în caz contrar se va executa o operaţie de scriere. Când un dizpozitiv **Slave** detectează că este adresat, atunci ar trebui să confirme adresarea setând linia **SDA** pe 0 logic **(LOW)** în al nouălea ciclu **SCL**.

Dacă nodul **Slave** adresat este ocupat sau nu poate răspunde la cererea dispozitivului **Master**, atunci linia **SDA** rămâne pe nivelul 1 logic **(HIGH)** în ciclul de confirmare a **SCL**. În acest caz **Master-ul** trimite o condiţie de **STOP** sau una de **REPEATED START** pentru a reiniţia transmisia. Un pachet de adresă constituit din adresa unui **Slave** şi o cerere de **READ/WRITE** este denumit generic **SLA+R**, respective **SLA+W**.

Bitul cel mai semnificativ al pachetului de adresare este transmis mai întâi. Adresele dispozitivurilor **Slave** sunt alocate, deobicei, de către utilizator (**atenție**, modulele de tip **Slave**, precum o memorie **EEPROM**, impun anumite restricții asupra adreselor care pot fi alese, deci şi numărul de dispozitive de acelaşi fel dintr-o rețea!), excluzând adresa **0000** ce este **rezervată** pentru un apel general. Atunci când un apel general este emis, toate dispozitivele **Slave** trebuie să răspundă prin setarea liniei **SDA** pe nivelul logic 0 **(LOW)** în timpul ciclului de recunoaştere. Atunci când după apelul general se transmite un bit de comandă pentru scriere, toate dispozitivurile **Slave** răspund prin setarea liniei **SDA** pe **LOW** în timpul ciclului de confirmare. Astfel, pachetele de date care vor urma vor fi recepţionate de către toate dispozitivurile **Slave** care au confirmat apelul general. În cazul în care apelul general este urmat de un bloc de read, atunci acest apel este ignorat de către sistem, deoarece acest lucru ar însemna ca toate dispozitivurile **Slave** ar transmite mesaje diferite concomitent.

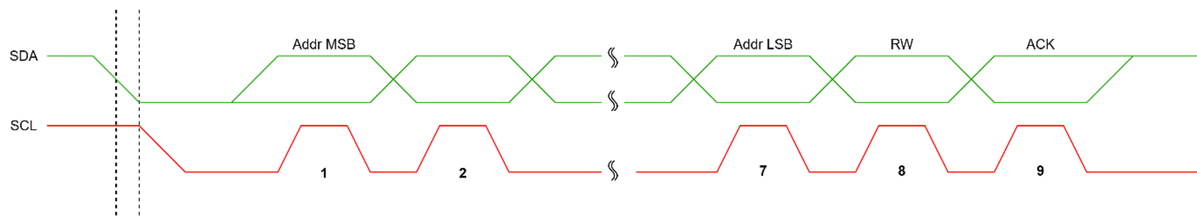


Figura 10.11 – Formatul pachetelor de adrese

Pachetele de date transmise pe bus-ul **TWI** au lungimea fixă de **9 biţi**, acestea fiind alcătuite dintr-un bit de **confirmare** şi **8 biţi de date**. Atunci când este iniţiat un transfer de date, dispozitivul **Master** generează pulsul de clock (condiţiile de **START** şi de **STOP**), în acest timp dispozitivul receiver confirmând datele recepţionate. Dispozitivul care recepţionează mesajul confirmă primirea acestuia setând linia **SDA** pe nivelul logic 0 **(LOW)** în timpul celui de al nouălea puls al liniei SCL. Dacă receiver-ul lasă linia SDA pe logic 1 **(HIGH)**, este semnalat cazul de **NACK** (No Acknowledge).

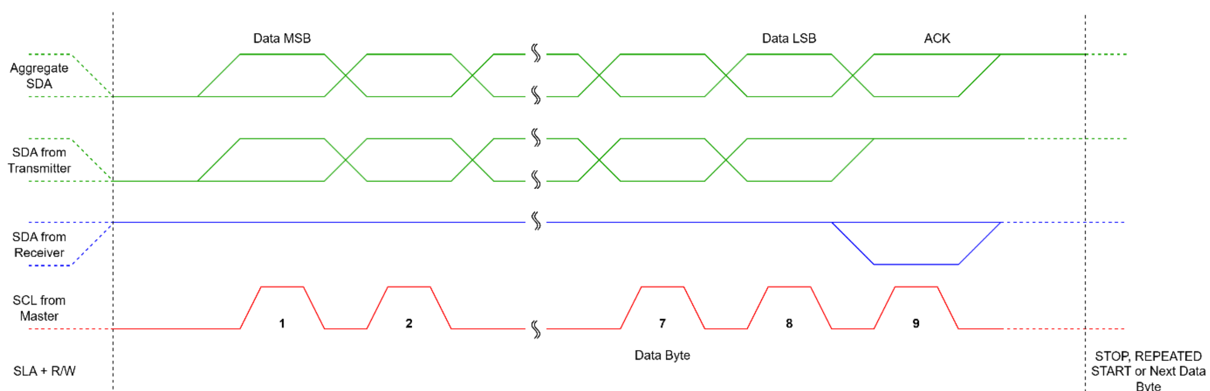


Figura 10.12 – Formatul pachetelor de adrese

10.11.4 COMBINAREA ADRESELOR ÎN TIMPUL TRANSMISIEI

O transmisie constă dintr-o condiţie de **START**, un pachet de tipul SLA+R/W, unul sau mai multe pachete de date şi o condiţie de **STOP**. Un mesaj gol, format dintr-un **START** urmat de o condiţie de **STOP**, este incorect.

Slave-ul poate extinde perioada **LOW** a SCL mutând linia SCL pe un nivel logic 0 **(LOW)**. Acest lucru este folositor dacă viteza clock-ului setată de dispozitivul **Master** este prea mare pentru **Slave** sau dacă **Slave-ul** are nevoie de mai mult timp pentru procesare între transmisiile de date. Faptul că **Slave-ul** măreşte perioada de nivel logic **LOW** a SCL-ului nu va afecta perioada **HIGH** a SCL-ului, aceasta fiind determinată de nodul **Master**. Ca o consecinţă, **Slave-ul** poate reduce viteza de transfer a datelor TWI prelungind ciclul SCL.

10.11.5 PINII SCL ŞI SDA

Aceşti pini au rolul de interfaţa între modulele TWI şi MCU. Contrar protocolului TWI, driverele de output nu conţin componente limitatoare pentru slew-rate sau filtrare de zgomot la intrare!

10.11.6 UTILIZAREA MODULULUI TWI

Interfața TWI din AVR este orientată pe octeți și bazată pe întreruperi. O întrerupere este generată după fiecare eveniment de pe magistrală, cum ar fi recepția unui octet sau transmiterea unei condiții **START**. Fiind bazată pe întreruperi, software-ul aplicației este liber să desfăşoare și alte operații în timpul unui transfer de octet prin TWI.

De reținut că bitul TWI Interrupt Enable **(TWIE)** din registrul **TWCR**, împreună cu bitul Global Interrupt Enable din **SREG**, permit aplicației să decidă dacă setarea flag-ului **TWINT** ar trebui sau nu să declanşeze o cerere de întrerupere. Dacă bitul **TWIE** este șters, aplicația trebuie să verifice (polling) flag-ul **TWINT** pentru a detecta acțiunile de pe magistrala TWI.

Atunci când flag-ul **TWINT** este setat, TWI a încheiat o operație și așteaptă răspunsul aplicației. În acest caz, registrul de stare TWI **(TWSR)** conține o valoare ce indică starea curentă a magistralei TWI. Software-ul aplicației poate decide apoi cum ar trebui să se comporte TWI în următorul ciclu al magistralei, prin manipularea registrelor **TWCR** și **TWDR**.

10.12 DESCRIEREA REGIŞTRILOR TWI

10.12.1 TWBR – REGISTRUL DE BIT RATE

Bit	7	6	5	4	3	2	1	0	
(0XB8)	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0	TWBR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Valoarea inițială	0	0	0	0	0	0	0	0	

Figura 10.13 – Registrul TWBR

- **Biții 7:0: Registrul de Bit Rate TWI**
  TWBR selectează factorul de divizare pentru generatorul de **bit rate**.

Generatorul de bit rate este un divizor de frecvență care generează frecvența ceasului **SCL** în modurile **Master**. Această unitate controlează perioada semnalului **SCL** atunci când funcționează în modul **Master**. Perioada semnalului **SCL** este determinată de valorile setate în Registrul de Rată de Bit TWI **(TWBR)** și de biții de prescaler din Registrul de Stare TWI **(TWSR)**. Funcționarea în modul **Slave** nu depinde de setările ratei de bit sau ale prescalerului, însă frecvența ceasului CPU în modul **Slave** trebuie să fie de **cel puțin 16 ori mai mare** decât frecvența **SCL**. Este de reținut că dispozitivele **Slave** pot prelungi perioada în care **SCL** este menținut pe nivel logic 0 **(LOW)**, reducând astfel perioada medie a ceasului pe magistrala TWI. Frecvența **SCL** este generată conform următoarei ecuații:

$$\text{frecvența SCL} = \frac{\text{frecvența clock-ului CPU}}{16+2(\text{TWBR}) * 4^{\text{TWPS}}}\text{, unde:}$$

- **TWBR** = valoarea registrului de bit rate **TWI**
- **TWPS** = valoarea biților de prescaler din registrul de status **TWI (TWSR)**

10.12.2 TWCR – REGISTRUL DE CONTROL

Bit	7	6	5	4	3	2	1	0	
(0XBC)	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE	TWCR
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W	
Valoarea inițială	0	0	0	0	0	0	0	0	

Figura 10.14 – Registrul TWCR

Registrul **TWCR** este utilizat pentru controlul funcționării interfeței TWI. Acesta este folosit pentru:

- activarea TWI
- Inițierea unei transmisii în modul **Master** prin aplicarea unei condiții **START** pe magistrală
- generarea unui semnal de confirmare (**ACK**) în modul **Receiver**
- generarea unei condiții **STOP**
- controlarea opririi magistralei în timp ce datele ce urmează a fi transmise sunt scrise în **TWDR**.

- **Bitul 7 – TWINT: Indicator De Întrerupere TWI**
  Acest bit este setat de hardware atunci când interfața TWI și-a încheiat sarcina curentă și așteaptă un răspuns din partea software-ului aplicației. Dacă bitul **I** din registrul **SREG** și bitul **TWIE** din registrul **TWCR** sunt setați, MCU-ul va sări la vectorul de întrerupere TWI. Cât timp indicatorul **TWINT** este setat, perioada în care semnalul **SCL** este pe nivel logic LOW este prelungită. Indicatorul **TWINT** trebuie șters de software prin scrierea valorii logice „1” în el.

Notă: indicatorul **TWI** nu este șters automat de hardware la executarea rutinei de întrerupere. De asemenea, ștergerea acestui indicator declanșează pornirea operației TWI, așa că toate accesările către registrul de adresă TWI (**TWAR**), registrul de stare TWI (**TWSR**) și registrul de date TWI (**TWDR**) trebuie finalizate înainte de ștergerea acestuia.

- **Bitul 6 – TWEA: Bit de activare a confirmării TWI**
  Bitul **TWEA** controlează generarea impulsului de confirmare **(ACK)**. Dacă **TWEA** este setat la 1, impulsul ACK este generat pe magistrala TWI dacă sunt îndeplinite următoarele condiții:
  1. A fost recepționată adresa proprie de **Slave** a dispozitivului.
  2. A fost recepționat un apel general, în timp ce bitul **TWGCE** din **TWAR** este setat.
  3. A fost recepționat un octet de date în modul **Master Receiver** sau **Slave Receiver**.
  Prin scrierea valorii 0 în **TWEA**, dispozitivul poate fi deconectat temporar, la nivel logic, de pe magistrala serială pe 2 fire. Recunoașterea adreselor poate fi reluată prin setarea din nou a bitului **TWEA** la 1.

- **Bitul 5 – TWSTA: Bit pentru condiția START TWI**
  Aplicația scrie valoarea 1 în **TWSTA** atunci când dorește ca dispozitivul să devină **Master** pe magistrala serială pe 2 fire. Hardware-ul TWI verifică dacă magistrala este liberă și generează o condiție **START** pe magistrală dacă aceasta este liberă. Dacă magistrala nu este liberă, TWI așteaptă până este detectată o condiție **STOP**, apoi generează o nouă condiție **START** pentru a revendica statutul de **Master** pe magistrală. Bitul **TWSTA** trebuie șters (setat la 0) de software după ce condiția **START** a fost transmisă.

- **Bitul 4 – TWSTO: Bit pentru condiția STOP TWI**
  Scrierea valorii 1 în bitul **TWSTO** în modul **Master** va genera o condiție **STOP** pe magistrala serială pe 2 fire. După ce condiția **STOP** este executată pe magistrală, bitul **TWSTO** este șters automat. În modul **Slave**, setarea bitului **TWSTO** poate fi folosită pentru recuperarea dintr-o stare de eroare. În acest caz, nu va fi generată o condiție **STOP**, dar interfața TWI va reveni într-un mod **Slave** bine definit, neadresat, și va elibera liniile **SCL** și **SDA** (trecându-le în stare de impedanță ridicată).

- **Bitul 3 – TWWC: Indicator de coliziune la scriere TWI**
  Bitul **TWWC** este setat atunci când se încearcă scrierea în registrul de date **(TWDR)** în timp ce bitul **TWINT** este pe nivel logic 0 **(LOW)**. Acest indicator se șterge prin scrierea în registrul **TWDR** atunci când **TWINT** este pe nivel logic 1 **(HIGH)**.

- **Bitul 2 – TWEN: Bit de activare TWI**
  Bitul **TWEN** activează funcționarea TWI și interfața TWI. Când este setat la 1, TWI preia controlul pinilor I/O conectați la **SCL** și **SDA**, activând limitatoarele slew-rate și filtrele de spike. Dacă acest bit este setat la 0, TWI este dezactivat și toate transmisiile TWI sunt întrerupte, indiferent dacă există sau nu o operație în desfășurare.

- **Bitul 1 – Res: Bit rezervat**
  Acest bit este rezervat și va fi citit întotdeauna ca 0.

- **Bitul 0 – TWIE: Bit de activare a întreruperilor TWI**
  Când acest bit este setat la 1 și bitul I din **SREG** este activat, cererea de întrerupere TWI va fi activă atât timp cât indicatorul **TWINT** este pe nivel logic 1 **(HIGH)**.

10.12.3 TWSR – REGISTRUL DE STATUS

Bit	7	6	5	4	3	2	1	0	
(0XB9)	TWS7	TWS6	TWS5	TWS4	TWS3	-	TWPS1	TWPS0	TWBR
Read/Write	R	R	R	R	R	R	R	R	
Valoarea inițială	1	1	1	1	1	0	0	0	

Figura 10.15 – Registrul TWSR

- **Biții 7:3 – TWS: Stare TWI**
  Acești 5 biți reflectă starea logicii TWI și a magistralei seriale pe 2 fire. Codurile de stare diferite sunt descrise mai târziu în această secțiune. Este de reținut că valoarea citită din **TWSR** conține atât valoarea pe 5 biți a stării, cât și valoarea pe 2 biți a prescalerului. Proiectantul aplicației ar trebui să mascheze biții prescaler la 0 atunci când verifică biții de stare. Astfel, verificarea stării devine independentă de setarea prescalerului. Această abordare este folosită în cazul în care nu este specificat altfel.

- **Bitul 2 – Res: Bit Rezervat**
  Acest bit este rezervat și va fi citit întotdeauna ca „0".

- **Biții 1:0 – TWPS: Biți De Prescaler TWI**
  Acești biți pot fi citiți și scriși și controlează prescalerul ratei de transfer pe bit.

10.12.4 TWDR – REGISTRUL DE DATE

Bit	7	6	5	4	3	2	1	0	
(0XBB)	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0	TWBR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Valoarea inițială	1	1	1	1	1	1	1	1	

Figura 10.16 – Registrul TWDR

În modul Transmitere, registrul **TWDR** conține următorul octet ce urmează a fi transmis. În modul Recepție, **TWDR** conține ultimul octet recepționat. Acesta poate fi scris atâta timp cât interfața TWI nu se află în procesul de transfer al unui octet. Această situație apare atunci când **indicatorul de întrerupere** TWI **(TWINT)** este setat de hardware.

Este de menționat că registrul de date nu poate fi inițializat de utilizator înainte ca prima întrerupere să aibă loc. Datele din **TWDR** rămân stabile atât timp cât **TWINT** este setat. În timpul transmiterii datelor, în paralel cu deplasarea datelor pe magistrală spre exterior, sunt citite simultan date de pe magistrală. **TWDR** conține întotdeauna ultimul octet prezent pe magistrală, cu excepția cazului în care dispozitivul iese dintr-un mod de economisire a energiei printr-o întrerupere TWI, caz în care conținutul **TWDR** este nedefinit. În cazul pierderii arbitrajului pe magistrală, nu se pierd date la trecerea din modul **Master** în modul **Slave**. Gestionarea bitului **ACK** este realizată automat de logica TWI, procesorul neavând acces direct la acest bit.

10.12.5 TWAR – REGISTRUL DE ADRESE (SLAVE)

Bit	7	6	5	4	3	2	1	0	
(0XBA)	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE	TWAR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Valoarea inițială	1	1	1	1	1	1	1	0	

Figura 10.17 – Registrul TWAR

Registrul **TWAR** trebuie încărcat cu adresa **Slave** pe 7 biți **(MSB-first)** la care interfața TWI va răspunde atunci când este programată ca **Slave Transmitter** sau **Receiver**, aceasta nefiind necesară în modurile **Master**. În sistemele cu mai mulți masteri, **TWAR** trebuie setat în masterii care pot fi adresați ca **Slave** de către alți masteri.

Cel mai puțin semnificativ bit **(LSB)** din **TWAR** este folosit pentru a activa recunoașterea adresei de apel general **(0x00)**. Există un comparator de adresă asociat care verifică prezența adresei **Slave** (sau a adresei de apel general, dacă este activată) în adresa serială recepționată. Dacă se găsește o potrivire, este generată o cerere de întrerupere.

- **Biții 7:1 – TWA: Registrul de Adresă TWI (Slave)**
  Acești biți constituie adresa Slave a unității TWI.

- **Bitul 0 – TWGCE: Bit de Activare a Recunoașterii Apelului General TWI**
  Dacă este setat, acest bit activează recunoașterea unui apel general transmis prin magistrala serială cu 2 fire (2-wired Serial Bus).

10.12.6 TWAMR – REGISTRUL DE ADRESE MASCĂ (SLAVE)

Bit	7	6	5	4	3	2	1	0	
(0XBD)	TWAM [6:0]							-	TWAMR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	
Valoarea inițială	0	0	0	0	0	0	0	0	

Figura 10.18 – Registrul TWAMR

- **Biții 7:1 – TWAM: Adresele Mască TWI**
  Registrul **TWAMR** poate fi încărcat cu o mască de adresă **Slave** pe 7 biți. Fiecare dintre biții din **TWAMR** poate masca (dezactiva) bitul de adresă corespunzător din registrul de adrese TWI (**TWAR**). Dacă bitul de mască este setat la 1, logica de potrivire a adresei ignoră comparația dintre bitul de adresă recepționat și bitul corespunzător din **TWAR**. **Figura 1.19** prezintă în detaliu logica de potrivire a adresei.

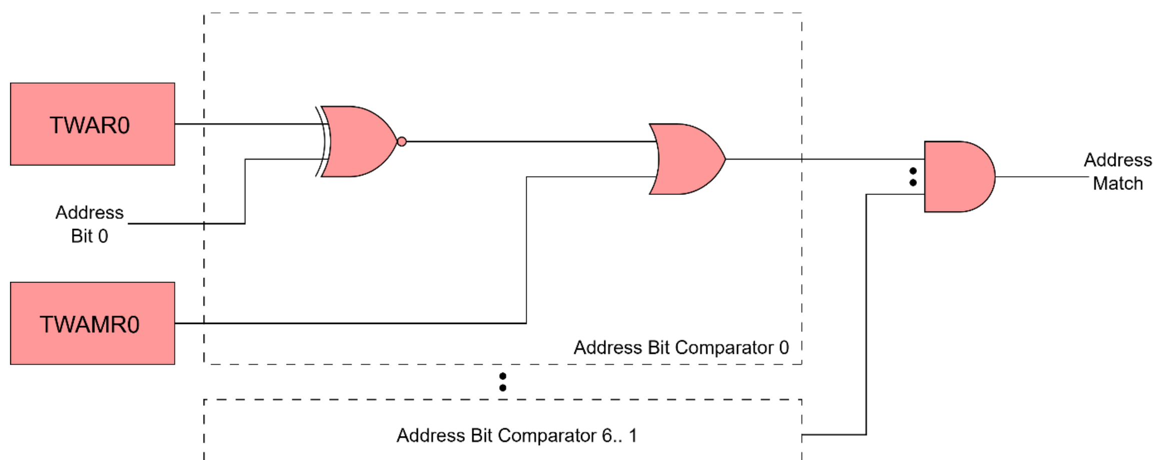


Figura 10.19 – Diagrama bloc TWI a logicii de potrivire a adreselor

10.13 PROBLEME

Pentru aplicațiile **I²C**, se pun la dispoziție următoarele biblioteci (headere) și funcțiile aferente:

`i2c.h`

Header-ul **`i2c.h`** declară funcțiile necesare pentru comunicarea **I²C** prin modulul TWI al microcontrolerului ATmega1280, permițând inițializarea interfeței (**`i2c_init()`**), generarea condițiilor **START** și **STOP** (**`i2c_start()`**, **`i2c_stop()`**), transmiterea unui octet către dispozitivul slave (**`i2c_write()`**) și citirea unui octet cu **ACK** sau **NACK** (**`i2c_read_ack()`**, **`i2c_read_nack()`**), oferind astfel o separare clară între interfața publică și implementarea efectivă.

```
/*------------------------------------------------------------------------
 * Fișier: i2c.h
 * Utilizat pentru declararea funcțiilor și interfeței I2C
 * (non-blocant, bazat pe întreruperi)
 *----------------------------------------------------------------------*/

#ifndef I2C_H
#define I2C_H

// Includes
#include <stdint.h>
#include <inavr.h>

/*------------------------------------------------------------------------
 * Public defines
 *----------------------------------------------------------------------*/

#ifndef F_CPU
#define F_CPU 16000000UL
#endif

// Delay specializat
#define DELAY_MS(ms) __delay_cycles((F_CPU / 1000UL) * (ms))

// Frecvența ceasului I2C (100 kHz)
#define SCL_CLOCK 100000UL

// Direcția tranzacției: scriere către Slave
#define TW_WRITE 0

/*
 * Status-ul tranzacției I2C
 * Enumerarea codurilor de stare pentru tranzacțiile I2C.
 * Utilă pentru depanare și gestionarea non-blocantă a transferurilor.
 */
typedef enum {
    I2C_STATUS_SUCCESS, // Tranzacția a fost pornită cu succes
    I2C_STATUS_BUSY,    // Magistrala este ocupată cu o altă tranzacție
    I2C_STATUS_ERROR    // Eroare pe magistrală (ex: NACK primit)
} i2c_status_t;

/*------------------------------------------------------------------------
 * Public (exported) functions
 *----------------------------------------------------------------------*/
```

```
/*
 * Funcția inițializează modulul TWI/I2C al microcontrollerului.
 * Configurează viteza și activează întreruperile pentru operare non-
 * blocantă.
 */
void i2c_init(void);

/*
 * Funcția pornește o tranzacție de scriere non-blocantă către un
 * dispozitiv Slave.
 */
i2c_status_t i2c_write_transaction(uint8_t address, uint8_t* data,
uint8_t length);


// Funcția verifică dacă o tranzacție este în desfășurare.
uint8_t i2c_is_busy(void);

/*
 * Funcția returnează ultimul status al tranzacției I2C.
 * Util pentru diagnostic și depanare.
 */
i2c_status_t i2c_get_last_status(void);

#endif
```

`i2c.c`

Fișierul `i2c.c` conține implementarea funcțiilor declarate în `i2c.h`, oferind suport pentru **comunicația I²C** prin modulul **TWI** al microcontrolerului **ATmega1280**. Funcția `i2c_init()` configurează modulul **TWI** ca **Master**, setând prescaler-ul și viteza magistralei **SCL**, astfel încât transferul de date să se realizeze la frecvența dorită (de exemplu **100 kHz** sau **400 kHz**).

Spre deosebire de implementările blocante, operațiile de citire și scriere se efectuează prin **tranzacții non-blocante**, inițiate cu funcțiile `i2c_write_transaction()` și `i2c_read_transaction()`. Aceste tranzacții permit microcontrolerului să continue executarea altor operații în timp ce transferul pe magistrala **I²C** se desfășoară, fiind gestionat complet de rutina de întrerupere **TWI**.

Starea magistralei și progresul fiecărei tranzacții pot fi monitorizate prin funcțiile `i2c_is_busy()` și `i2c_get_last_status()`. În plus, codurile de stare din registrul **TWSR** sunt utilizate intern pentru a asigura o comunicare fiabilă cu dispozitivele slave, tratând corect răspunsurile **ACK/NACK** și situațiile de eroare.

Această arhitectură permite un control precis al magistralei **I²C fără blocarea procesorului**, fiind ideală pentru aplicații embedded care necesită multitasking sau operarea mai multor periferice simultan.

```
/*----------------------------------------------------------------------
 * Fișier: i2c.c
 * Implementarea funcțiilor și rutinelor asociate interfeței I2C/TWI
 * (non-blocant, bazat pe întreruperi)
 *--------------------------------------------------------------------*/

// Includes
#include "i2c.h"
#include <ioavr.h>
#include <inavr.h>

/*----------------------------------------------------------------------
 * Public defines
 *--------------------------------------------------------------------*/
```

```
// Condiție START transmisă
#define TW_START                  0x08
// Master Transmit: adresa + W transmisă, ACK primit
#define TW_MT_SLA_ACK             0x18
// Master Transmit: octet de date transmis, ACK primit
#define TW_MT_DATA_ACK            0x28
// Master Receive: adresa + R transmisă, ACK primit
#define TW_MR_SLA_ACK             0x40
// Master Receive: octet de date recepționat, ACK returnat
#define TW_MR_DATA_ACK            0x50
// Master Receive: octet de date recepționat, NACK returnat
#define TW_MR_DATA_NACK           0x58

/*
 * Reprezintă starea curentă a modulului I2C/TWI.
 * Utilizată pentru a coordona transferurile non-blocante.
 */
typedef enum {
    I2C_STATE_IDLE,     // Nicio tranzacție în desfășurare
    I2C_STATE_WRITING,  // Se efectuează o tranzacție de scriere
    I2C_STATE_READING   // Se efectuează o tranzacție de citire
} i2c_state_t;

/*--------------------------------------------------------------------------
 * Global variables
 *------------------------------------------------------------------------*/

// Starea curentă
static volatile i2c_state_t g_i2c_state = I2C_STATE_IDLE;

// Ultimul status returnat
static volatile i2c_status_t g_i2c_status = I2C_STATUS_SUCCESS;

static uint8_t g_slave_address;   // Adresa slave (cu bitul R/W inclus)
static uint8_t* g_data_ptr;       // Pointer către bufferul de date
static uint8_t g_data_len;        // Lungimea totală a transferului
static uint8_t g_data_idx;        // Indexul curent în buffer

/*--------------------------------------------------------------------------
 * Public functions (exported from .h)
 *------------------------------------------------------------------------*/

/*
 * Funcția inițializează modulul TWI/I2C.
 * - Configurează prescaler-ul și frecvența de lucru (SCL_CLOCK).
 * - Activează perifericul TWI (fără a activa întreruperile încă).
 */
void i2c_init(void) {
    TWSR = 0x00; // Prescaler = 1
    // Baud-rate generator pentru SCL
    TWBR = ((F_CPU / SCL_CLOCK) - 16) / 2;
    // Activează TWI, fără întreruperi
    TWCR = (1 << TWEN);
}

/*
 * Funcția verifică dacă magistrala I2C este ocupată.
 * Returnează:
 *   - 1 dacă o tranzacție este în desfășurare
 *   - 0 dacă modulul este inactiv (IDLE)
 */
```

```
uint8_t i2c_is_busy(void) {
    return (g_i2c_state != I2C_STATE_IDLE);
}

/*
 * Funcția returnează ultimul status înregistrat al tranzacției.
 * Poate fi folosită pentru depanare.
 */
i2c_status_t i2c_get_last_status(void) {
    return g_i2c_status;
}

/*
 * Funcția pornește o tranzacție de scriere NON-BLOCANTĂ.
 * Returnează:
 *   - I2C_STATUS_SUCCESS dacă tranzacția a fost pornită
 *   - I2C_STATUS_BUSY dacă magistrala era deja ocupată
 */
i2c_status_t i2c_write_transaction(uint8_t address, uint8_t* data,
                                   uint8_t length) {
    if (g_i2c_state != I2C_STATE_IDLE) {
        return I2C_STATUS_BUSY;
    }
    g_i2c_state = I2C_STATE_WRITING;
    g_i2c_status = I2C_STATUS_BUSY;
    g_slave_address = (address << 1) | TW_WRITE; // Adresa + bit W
    g_data_ptr = data;
    g_data_len = length;
    g_data_idx = 0;

    // Lansăm tranzacția prin generarea condiției START + întrerupere
    TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN) | (1 << TWIE);

    return I2C_STATUS_SUCCESS;
}

/*
 * Funcția pornește o tranzacție de citire NON-BLOCANTĂ.
 * Returnează:
 *   - I2C_STATUS_SUCCESS dacă tranzacția a fost pornită
 *   - I2C_STATUS_BUSY dacă magistrala era deja ocupată
 */
i2c_status_t i2c_read_transaction(uint8_t address, uint8_t* data,
                                  uint8_t length) {
    if (g_i2c_state != I2C_STATE_IDLE) {
        return I2C_STATUS_BUSY; // Magistrala e ocupată
    }
    g_i2c_state = I2C_STATE_READING;
    g_i2c_status = I2C_STATUS_BUSY;
    g_slave_address = (address << 1) | 1; // Adresa + bit R
    g_data_ptr = data;
    g_data_len = length;
    g_data_idx = 0;

    // Pentru citire, inițiem cu un START
    TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN) | (1 << TWIE);

    return I2C_STATUS_SUCCESS;
}
```

```
/*
 * Această rutină implementează logica mașinii de stări pentru TWI/I2C.
 * Gestionează atât operațiile de scriere, cât și pe cele de citire,
 * pe baza valorii registrului TWSR (status).
 */
#pragma vector = TWI_vect
__interrupt void TWI_ISR(void) {
    switch (TWSR & 0xF8) { // Se maschează doar biții de status
        case TW_START: // Condiție START trimisă
            TWDR = g_slave_address; // Se trimite adresa + bit R/W
            TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWIE);
            break;

        // Slave a răspuns cu ACK după adresă
        case TW_MT_SLA_ACK:

        // Slave a răspuns cu ACK după un octet de date
        case TW_MT_DATA_ACK:
            if (g_data_idx < g_data_len) {
                // Mai sunt date de transmis
                TWDR = g_data_ptr[g_data_idx++];
                TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWIE);


            } else {
                // S-a terminat transmisia
                TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);
                g_i2c_status = I2C_STATUS_SUCCESS;
                g_i2c_state = I2C_STATE_IDLE;
            }
            break;
        // S-a trimis adresa + R, Slave a răspuns cu ACK
        case TW_MR_SLA_ACK:
            // Dacă mai este mai mult de 1 octet de citit e returnat ACK
            if (g_data_len - g_data_idx > 1) {
           TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWIE) | (1 << TWEA);
            } else {
                // Ultimul octet, deci se returnează NACK
                TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWIE);
            }
            break;

        case TW_MR_DATA_ACK: // S-a citit un octet și s-a trimis ACK
            g_data_ptr[g_data_idx++] = TWDR;
            if (g_data_len - g_data_idx > 1) {
                // Mai sunt date de citit, deci se cere cu ACK
           TWCR = (1 << TWINT)|(1 << TWEN)|(1 << TWIE) | (1 << TWEA);
            } else {
                // Urmează ultimul octet, deci se cere fără ACK
                TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWIE);
            }
            break;

        // S-a citit ultimul octet și s-a trimis NACK
        case TW_MR_DATA_NACK:
            g_data_ptr[g_data_idx++] = TWDR;
            TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN); // STOP
            g_i2c_status = I2C_STATUS_SUCCESS;
            g_i2c_state = I2C_STATE_IDLE;
            break;
```

```
        default:
            g_i2c_status = I2C_STATUS_ERROR; // Se semnalează eroarea
            // STOP - se eliberează magistrala
            TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);
            g_i2c_state = I2C_STATE_IDLE;
            break;
    }
}
```

10.13.1 CITIREA ȘI SCRIEREA ÎN EEPROM PRIN I²C

Cerință: Să se realizeze un program care să exemplifice utilizarea protocolului I²C prin interacțiunea directă cu memoria **EEPROM FT24C08A**. Programul trebuie să asigure comunicarea **Master-Slave** prin controlul registrelor **TWI**, implementând operații complete de **citire** și **scriere**.

Sugestii:

- *Se pot folosi secvențele de **START**, **STOP** și **REPEATED START** pentru a inițializa și încheia corect tranzacțiile pe bus.*
- *Dispozitivul Slave se selectează prin trimiterea pachetului de adresă corespunzător, iar comunicația trebuie gestionată atât în modul **Master-Transmitter**, cât și în **Master-Receiver**.*
- *Pachetele de date de 8 biți trebuie transmise și recepționate conform specificațiilor protocolului I²C.*
- *Este necesară monitorizarea **biților de ACKNOWLEDGE** în registrul TWSR pentru confirmarea reușitei fiecărei transmisii și pentru a evita erorile de comunicare.*
- *Pentru validare, se poate implementa o operație de tip **citire–modificare–scriere** asupra memoriei **EEPROM**, verificând astfel atât funcționalitatea transmisiei, cât și consistența datelor.*

`main.c`

Fișierul **`main.c`** configurează interfața **TWI** a microcontrolerului pentru a comunica pe magistrala **I²C** cu memoria **EEPROM FT24C08A**. Se implementează o mașină de stări care realizează secvențe complete de **citire–modificare–scriere**: inițiere cu **START** și adresare, transmiterea adresei interne, citirea unui octet, incrementarea valorii și scrierea acesteia înapoi în memorie. Protocolul este controlat direct prin registre, verificând **ACK-urile** și gestionând atât **scrierea**, cât și **citirea** cu secvențe de **START**, **REPEATED START** și **STOP**.

```
/*-----------------------------------------------------------------------
 * Fișier: main.c
 * Utilizat la implementarea aplicației de citire / scriere în EEPROM
 *---------------------------------------------------------------------*/

// Includes
#include <ioavr.h>
#include <inavr.h>
#include "i2c.h"

/*-----------------------------------------------------------------------
 * Public defines
 *---------------------------------------------------------------------*/

#ifndef F_CPU
#define F_CPU 16000000UL
#endif

// Adresa de bază a EEPROM-ului pe magistrala I2C
#define EEPROM_DEVICE_ADDR 0x50
```

```
// Stările mașinii de stări a aplicației
typedef enum {
    STATE_IDLE,             // Nu se face nicio operație
    STATE_START_READ,       // Se inițiază citirea
    STATE_WRITE_ADDR_READ,  // S-a scris adresa, urmează citirea
    STATE_READ_DATA,        // Se citește efectiv datele din EEPROM
    STATE_START_WRITE,      // Se pregătește secvența de scriere în EEPROM
    STATE_WRITE_ADDR_WRITE,// Se scrie adresa în care se vor salva datele
    STATE_WRITE_DATA,       // Se scriu efectiv datele
    STATE_STOP              // Finalizarea tranzacției
} app_state_t;

// Variabilă globală volatilă pentru mașina de stări
volatile app_state_t g_app_state = STATE_IDLE;

// Buffer pentru datele transmise
uint8_t g_data_buffer[3];

// Variabilă unde se salvează valoarea din EEPROM
uint8_t g_read_data = 0;

// Adresa internă EEPROM pentru citire / scriere
uint8_t g_eeprom_mem_addr = 0x20;

int main(void) {
    i2c_init();              // Se inițializează perifericul TWI (I2C)
    __enable_interrupt();    // Se activează întreruperile globale

    g_app_state = STATE_START_READ; // Se pornește cu o citire din EEPROM

    while (1) {
        if (!i2c_is_busy()) {
            switch (g_app_state) {
                case STATE_START_READ:
    // Pregătim bufferul pentru a transmite adresa internă a EEPROM-ului
                    g_data_buffer[0] = (EEPROM_DEVICE_ADDR << 1) | 0;
    // Adresa internă din EEPROM
                    g_data_buffer[1] = g_eeprom_mem_addr;
    // Lansăm tranzacția de scriere (se trimite adresa internă)
                    if (I2C_STATUS_SUCCESS ==
i2c_write_transaction(EEPROM_DEVICE_ADDR, g_data_buffer, 2)) {
                        g_app_state = STATE_WRITE_ADDR_READ;
// După ce s-a scris adresa, se trece la citire
                    }
                    break;
/*
 * După ce s-a setat adresa internă, se face un REPEATED START pentru a
 * citi din EEPROM
 */
                case STATE_WRITE_ADDR_READ:
                    // Adresa dispozitivului + bit de citire
                    g_data_buffer[0] = (EEPROM_DEVICE_ADDR << 1) | 1;

                   // Se lansează tranzacția de citire
                   if (I2C_STATUS_SUCCESS ==
i2c_write_transaction(EEPROM_DEVICE_ADDR, g_data_buffer, 1)) {
                        // Urmează citirea datelor
                        g_app_state = STATE_READ_DATA;
                    }
                    break;
```

```
                    case STATE_READ_DATA:
                        g_read_data = g_data_buffer[0];
                        g_read_data++;
                        g_app_state = STATE_START_WRITE;
                        break;

                    case STATE_START_WRITE:
                        g_data_buffer[0] = (EEPROM_DEVICE_ADDR << 1) | 0;
                        g_data_buffer[1] = g_eeprom_mem_addr
                        g_data_buffer[2] = g_read_data


                        if (I2C_STATUS_SUCCESS ==
i2c_write_transaction(EEPROM_DEVICE_ADDR, g_data_buffer, 3)) {
                             g_app_state = STATE_STOP;
                        }
                        break;
                    // S-a terminat ciclul citire -> modificare -> scriere
                    case STATE_STOP:

                        g_app_state = STATE_IDLE;
                        break;

                    case STATE_IDLE:

// În stare de idle, se reia ciclul (se face din nou citire)
                    default:
                        g_app_state = STATE_START_READ;
                        break;
                }
            }
        }
}
```

10.14 BIBLIOGRAFIE

1. *"8-bit AVR Microcontroller ATmega 1280 Datasheet", Microchip Technology*
2. *"Schematic for UNI Clicker", Mikroe*
3. *"Schematic for ATmega1280: SiBrain", Mikroe*
4. *"Inter-Integrated Circuits", Wikipedia*
5. *"I²C Basics", Circuit Basics*