

Microcontrolere și Microprocesoare

Note de curs

Tiberiu T. Cociaș



Universitatea
Transilvania
din Brașov

2026

Cuprins

1	Introducere în sistemele cu microcontrolere	3
1.1	La ce servesc microcontrolerele	4
1.1.1	Robotică și mecatronică	4
1.1.2	Automobilul	4
1.1.3	Electronica de consum și aparatura casnică	5
1.1.4	Controlul mediului și instrumentația	5
1.2	Ce este un microcontroler	6
1.2.1	Resursele minime	6
1.2.2	Resursele opționale	7
1.3	Microprocesor, microcontroler și sistem pe cip	7
1.4	Sistemul încorporat și criteriile lui de proiectare	9
1.4.1	Determinismul	10
1.4.2	Timpul de răspuns	11
1.4.3	Consumul	11
1.4.4	Costul unitar	11
1.4.5	Fiabilitatea	12
1.5	Scurt istoric al microcontrolerelor	12
1.6	Anatomia unui microcontroler	14
1.6.1	Nucleul	15
1.6.2	Memoria de program	15
1.6.3	Memoria de date	16
1.6.4	Perifericele	16
1.6.5	Oscilatorul	17

1.6.6	Alimentarea, resetul și pinii	17
1.7	Microcontrolerul acestui curs: ATmega328P și platforma Arduino	18
1.8	Harta cărții	20
2	Arhitectura unui microcontroler	23
2.1	Bitul, octetul și citirea unei valori	24
2.1.1	Aceeași valoare, trei scrieri	24
2.2	Registrii	26
2.3	Registrii cu funcții speciale	26
2.4	Memoria unui microcontroler	28
2.4.1	Memorii volatile	29
2.4.2	Memorii nevolatile	29
2.5	Unitatea centrală de procesare	30
2.5.1	Unitatea aritmetică și logică	30
2.5.2	Registrul de stare	31
2.5.3	Unitatea de control	32
2.5.4	Stiva	32
2.6	Semnalele la interfața unității centrale cu exteriorul	33
2.7	Arhitectura generală a unui microcontroler	33
2.7.1	Arhitectura von Neumann	33
2.7.2	Arhitectura Harvard	34
2.7.3	Execuția cvasiparalelă și conceptul de pipeline	35
2.8	Setul de instrucțiuni	35
2.8.1	RISC față de CISC	36
2.9	Arhitectura ATmega328P	37
2.9.1	De ce SRAM-ul mic este constrângerea reală	37
2.10	Circuitul oscilator	39
2.10.1	Oscilatorul cu cristal de cuarț	39
2.10.2	Oscilatorul RC	39
2.10.3	Când contează precizia	40
2.11	Ciclul de instrucțiune	40
2.12	Alimentarea microcontrolerului	41
2.12.1	De ce condensatoarele de decuplare nu sunt opționale	42
2.13	Un exemplu de conectare	43
3	Programarea unui microcontroler	47
3.1	Ce înseamnă a programa un microcontroler	47
3.2	Etapele programării: de la codul sursă la biții din Flash	48
3.2.1	Preprocesarea	48
3.2.2	Compilarea propriu-zisă	50
3.2.3	Asamblarea	50

3.2.4	Editarea de legături	51
3.2.5	Conversia în format Intel HEX	52
3.2.6	Transferul în memorie și verificarea	52
3.3	Formatul fișierului Intel HEX	53
3.3.1	Structura unei linii	53
3.3.2	Suma de control	54
3.3.3	Un exemplu de la începutul unui program AVR	55
3.4	Structura memoriei Flash: bootloader și zona de aplicație	56
3.4.1	Cele două secțiuni ale memoriei Flash	56
3.4.2	Ce face un bootloader	57
3.4.3	Prețul plătit și ce se pierde dacă bootloader-ul dispare	57
3.5	Programarea modulelor Arduino	58
3.5.1	Resetarea automată	58
3.5.2	Dialogul dintre avrdude și bootloader	59
3.5.3	O placă Arduino în rol de programator	59
3.6	Programarea prin ISP cu programator dedicat	59
3.6.1	Când este necesar fiecare mod de programare	60
3.7	Limbaje de programare	61
3.7.1	Limbaajul de asamblare	61
3.7.2	Limbaajul C	62
3.7.3	Ce ascund bibliotecile Arduino	62
3.8	Depanarea unui microcontroler	64
3.8.1	Depanarea la nivel hardware: debugWIRE, JTAG, SWD	64
3.8.2	De ce mesajele pe serială rămân unele principale	65
3.9	Greșeli tipice și cum se recunosc	66
3.9.1	Memoria Flash plină	66
3.9.2	Memoria SRAM epuizată	66
3.9.3	Ceasul configurat altfel decât presupune codul	67
3.9.4	Alte confuzii frecvente	68
4	Porturi de intrare și ieșire	71
4.1	Porturile de intrare și ieșire	72
4.2	Portul digital	73
4.3	Arhitectura porturilor	73
4.3.1	Structura internă a unui pin	74
4.3.2	Cele trei registre ale unui port	75
4.4	Configurarea porturilor	76
4.4.1	Configurarea prin registre	76
4.4.2	Configurarea prin funcțiile Arduino	78
4.5	Transferul de date către microcontroler	79
4.6	Metastabilitatea	80

4.7	Circuitul sincronizator	81
4.8	Rezistențele de pull-up și de pull-down	82
4.8.1	Cum funcționează un pull-up	83
4.8.2	Pull-up-ul intern al AVR-ului și când nu este suficient	84
4.9	Pinii cu funcții multiple	85
4.10	Protecția pinilor digitali	86
4.10.1	Protecțiile interne	86
4.10.2	Curentul maxim pe pin și pe port	87
4.10.3	Rezistența de limitare pentru un LED	88
4.11	Circuitele conexe	89
4.11.1	Butoane și comutatoare	89
4.11.2	Relee	90
4.11.3	LED-uri și afișoare	91
4.11.4	Optocuploare	91
4.11.5	Ce reține proiectantul	92
5	Sistemul de întreruperi	95
5.1	Prezentare generală	95
5.1.1	Alternativa: interogarea ciclică	96
5.1.2	Analogia cu soneria	97
5.2	Caracteristici ale sistemului de întreruperi	98
5.3	Principiul de funcționare	99
5.3.1	Sucesiunea de operații	99
5.3.2	Latența răspunsului la întrerupere	100
5.4	Rutina de tratare a întreruperii	101
5.4.1	Salvarea și restaurarea contextului	101
5.4.2	Reguli de scriere a unei rutine de tratare	102
5.5	Tabelul vectorilor de întrerupere	103
5.5.1	Ce este și unde se află	103
5.5.2	Vectorii de întrerupere ai microcontrolerului ATmega328P	104
5.6	Surse de întrerupere	104
5.6.1	Întreruperi externe	106
5.6.2	Întreruperi interne	107
5.7	Priorități și imbricarea întreruperilor	107
5.7.1	Priorități hardware și priorități software	107
5.7.2	Cum stau lucrurile pe AVR	108
5.8	Variabile partajate și condiții de cursă	109
5.8.1	Optimizarea compilatorului și cuvântul cheie volatile	109
5.8.2	Condiția de cursă	110
5.9	Aplicație cu sistemul de întreruperi	112
5.9.1	Montajul	112

5.9.2	Inițializarea porturilor	112
5.9.3	Inițializarea sistemului de întreruperi	113
5.9.4	Rutina de tratare și activarea globală	114
5.9.5	Varianța Arduino	115
5.9.6	Problema oscilațiilor de contact	116
6	Module de tip timer	119
6.1	Prezentare generală	119
6.1.1	Timer sau counter	120
6.1.2	La ce folosește un timer	120
6.1.3	De ce funcția <code>delay()</code> este cea mai proastă soluție	121
6.2	Clasificarea modulelor de tip timer	122
6.2.1	Timerul simplu	122
6.2.2	Numărătorul de evenimente	123
6.2.3	Timerul generator de PWM	123
6.2.4	Temporizatorul de gardă	123
6.2.5	Ceasul de timp real	123
6.2.6	Timerele microcontrolerului ATmega328P	124
6.2.7	Modurile de lucru	125
6.3	Modalități de accesare a datelor din timer	126
6.3.1	Citirea directă a registrului de contor	126
6.3.2	Interogarea indicatorilor de stare	127
6.3.3	Folosirea întreruperilor	128
6.4	Utilizarea timerului cu întrerupere	129
6.4.1	Întreruperea la depășire	129
6.5	Circuitul de prescalare	130
6.5.1	Legătura dintre frecvență, prescalare și perioadă	130
6.6	Caracteristicile semnalelor periodice	133
6.6.1	De ce factorul de umplere este mărimea care contează	133
6.7	Generarea unui semnal cu factor de umplere variabil (PWM)	134
6.8	Implementarea PWM	135
6.8.1	PWM rapid	135
6.8.2	PWM cu fază corectă	136
6.8.3	Configurarea directă a registrelor	136
6.8.4	Legătura cu funcția <code>analogWrite()</code>	137
6.9	Capturarea evenimentelor externe	138
6.9.1	Mecanismul capturii	138
6.9.2	Configurarea unei măsurători	140
6.9.3	Aplicație: senzorul cu ultrasunete	140
6.9.4	Alte utilizări ale capturii	142
6.10	Concluzii	143

7	Convertorul analog-numeric	145
7.1	De ce este nevoie de conversie analog-numerică	145
7.2	Caracteristicile de performanță ale unui convertor	146
7.3	Etapele procesului de conversie	147
7.3.1	Eșantionarea	148
7.3.2	Reținerea și circuitul de eşantionare și menținere	148
7.3.3	Cuantizarea	149
7.3.4	Codificarea și stocarea	150
7.4	Rata de eşantionare și fenomenul de aliasing	150
7.4.1	Rata de eşantionare	150
7.4.2	Teorema eşantionării	151
7.4.3	Aliasing	151
7.4.4	Filtrul anti-alias	152
7.5	Rezoluția și eroarea de cuantizare	153
7.6	Tensiunea de referință	154
7.6.1	Sursele de referință ale microcontrolerului ATmega328P	155
7.7	Timpul de conversie	156
7.7.1	Ceasul convertorului din ATmega328P	157
7.8	Arhitecturi de implementare	158
7.8.1	Comparatorul, convertorul de un bit	158
7.8.2	Convertorul cu aproximări succesive	159
7.8.3	Convertorul paralel (flash)	161
7.8.4	Convertorul cu rampă	161
7.8.5	Convertorul sigma-delta	162
7.8.6	Compararea arhitecturilor	162
7.9	Erori de conversie	163
7.10	Modulul ADC al microcontrolerului ATmega328P	164
7.10.1	Registrul ADMUX	164
7.10.2	Registrul ADCSRA	164
7.10.3	Registrele de rezultat și ordinea de citire	164
7.10.4	Citirea cu funcțiile de bibliotecă	165
7.10.5	Aceeași citire scrisă direct pe registre	166
7.10.6	Transformarea valorii brute în tensiune	167
7.11	Greșeli tipice și recomandări practice	168
7.12	Concluzii	169
8	Comunicația serială asincronă	173
8.1	Sistemele de comunicație	173
8.2	Tipuri de comunicație	174
8.2.1	Serial față de paralel	174
8.2.2	Sincron față de asincron	175

8.2.3	Simplex, semiduplex, duplex integral	175
8.2.4	Punct la punct față de magistrală	176
8.3	Comunicația serială sincronă și asincronă	176
8.4	Modulul UART	176
8.5	Formatul cadrului	177
8.6	Sincronizarea între emițător și receptor	178
8.6.1	Eșantionarea la mijlocul bitului	179
8.6.2	Supraeșantionarea de 16 ori	179
8.6.3	De ce eroarea de ceas se acumulează	179
8.7	Calculul ratei de transfer	181
8.7.1	Formula registrului UBRR	181
8.7.2	Modul de viteză dublă	182
8.7.3	De ce există cristale de 14,7456 MHz	182
8.8	Registrele modulului USART al microcontrolerului ATmega328P	183
8.8.1	Registrul de date UDR0	183
8.8.2	Registrele de control și stare	184
8.8.3	Indicatorii de eroare	185
8.9	Aplicație: transmisia și recepția unui caracter	185
8.9.1	Inițializarea	185
8.9.2	Transmisia și recepția prin interogare	186
8.9.3	Echivalentul cu funcțiile Arduino	188
8.10	Recepția cu întrerupere și tamponul circular	188
8.11	De la UART la USART	191
8.12	Standardele RS-232 și RS-485	192
8.12.1	Nivelurile TTL/CMOS nu sunt RS-232	192
8.12.2	RS-485 și semnalul diferențial	193
8.12.3	Terminarea liniei	194
8.13	Concluzii	195
9	Magistralele sincrone: SPI și I²C	197
9.1	De ce este nevoie de o magistrală sincronă	197
9.2	Magistrala I ² C: prezentare generală	198
9.3	Principiul de funcționare al magistralei I ² C	199
9.3.1	Colectorul deschis și rezistențele de ridicare	200
9.3.2	Adresarea pe șapte biți	201
9.3.3	Condițiile de start și de stop	201
9.3.4	Confirmarea ACK și NACK	202
9.4	Tranzacții pe magistrala I ² C	203
9.4.1	Tranzacția de scriere	203
9.4.2	Tranzacția de citire	203
9.4.3	Arbitrajul și întinderea ceasului	204

9.5	Magistrala SPI: principiul de funcționare	205
9.5.1	Cele patru linii	205
9.5.2	Registrul de deplasare circular	206
9.6	Transferul datelor și modurile SPI	207
9.6.1	Polaritatea și faza ceasului	207
9.6.2	De ce un mod greșit nu produce niciun mesaj de eroare	208
9.7	Topologii SPI	209
9.8	Modulul SPI al microcontrolerului ATmega328P	210
9.8.1	Transferul unui octet prin acces direct la registre	211
9.9	Aplicații cu bibliotecile SPI și Wire	212
9.9.1	Biblioteca SPI	212
9.9.2	Biblioteca Wire	214
9.9.3	Scanarea magistralei I ² C	215
9.10	Comparație între UART, SPI și I ² C	217
9.11	Concluzii	218
10	Proiectarea și testarea aplicațiilor cu microcontrolere	221
10.1	Proiectare și testare	221
10.1.1	De ce eșuează un proiect	222
10.1.2	Cele două etape și ciclul dintre ele	222
10.2	Un exemplu de referință	223
10.3	Descrierea sistemului și caietul de sarcini	225
10.3.1	Ce înseamnă o cerință verificabilă	225
10.3.2	Cerințe funcționale și cerințe nefuncționale	225
10.4	Partiționarea sistemului	227
10.4.1	Interfața se stabilește înaintea implementării	227
10.4.2	Diagrama bloc a exemplului	228
10.5	Alegerea microcontrolerului	229
10.6	Detalierea subsistemelor	230
10.6.1	Bucula principală față de arhitectura condusă de evenimente	230
10.6.2	Mașina cu stări finite	230
10.7	Activitatea principală	234
10.8	Scrierea și testarea primară a codului	236
10.8.1	De ce nu se testează direct pe instalația reală	237
10.8.2	Cod ușor de testat	237
10.9	Planul de testare	238
10.9.1	Niveluri de testare	238
10.10	Depanarea sistematică	239
10.11	Documentarea	242
11	Studiu de caz: robotul GT	247

11.1	Contextul și tema de proiect	247
11.2	Caietul de sarcini	248
11.3	Partiționarea sistemului	250
11.4	Bugetul de resurse	251
11.4.1	Pinii	252
11.4.2	Timerele, canalele analogice și întreruperile	253
11.5	Arhitectura programului	254
11.5.1	Structura pe mașină de stări	255
11.6	Comanda motoarelor	258
11.7	Odometria din encodere	261
11.8	Urmărirea liniei	263
11.9	Planul de testare	266
11.10	Ce a mers și ce nu	268

1. Introducere în sistemele cu microcontrolere

- La ce servesc microcontrolerele
- Ce este un microcontroler
- Microprocesor, microcontroler și sistem pe cip
- Sistemul încorporat și criteriile lui de proiectare
- Scurt istoric al microcontrolerelor
- Anatomia unui microcontroler
- Microcontrolerul acestui curs: ATmega328P și platforma Arduino
- Harta cărții
- Întrebări de verificare

Cartea aceasta se ocupă de un obiect pe care aproape nimeni nu îl vede, deși fiecare dintre noi îl folosește de zeci de ori pe zi. Un calculator de birou sau un telefon sunt recunoscute imediat drept calculatoare: au un ecran, o tastatură, un sistem de operare. În schimb, mașina de spălat din baie, telecomanda televizorului, cheia de contact a automobilului, termostatul de pe perete, cântarul din bucătărie și frâna antiblocare a mașinii conțin, fiecare, cel puțin un calculator complet, cu procesor, memorie și program, numai că acest calculator nu se prezintă niciodată ca atare. El nu are ecran propriu, nu rulează aplicații, nu poate fi reprogramat de utilizator și, în mod normal, nu este observat decât atunci când se strică.

Obiectul acesta se numește *microcontroler*, iar sistemele construite în jurul lui se numesc *sisteme încorporate* (în engleză *embedded systems*), pentru că partea de calcul este încorporată într-un produs care, din punctul de vedere al cumpărătorului, nu este un calculator, ci o mașină de spălat, o bormașină sau o pompă de insulină. Scopul acestui prim capitol este să lămurească ce este un microcontroler, prin ce se deosebește el de procesorul unui calculator obișnuit, de ce un sistem încorporat se proiectează după alte criterii decât un calculator de birou și de ce, pentru a învăța toate acestea, vom folosi un microcontroler de opt biți vechi de peste cincisprezece ani.

Nu este nevoie de nicio experiență anterioară cu microcontrolere pentru a citi acest capitol. Se presupun cunoscute doar noțiunile elementare de electronică digitală, adică ce înseamnă un nivel logic, un registru și un bit, și noțiunile de bază de programare în C.

1.1 La ce servesc microcontrolerele

Prima întrebare pe care și-o pune un student de la o specializare de mecatronică sau de design de produs este de ce ar trebui să învețe programarea unui circuit integrat, câtă vreme meseria lui va fi să proiecteze un produs, nu un calculator. Răspunsul este că, de câteva decenii, aproape orice produs care se mișcă, se încălzește, măsoară ceva sau interacționează cu un om conține un microcontroler, iar comportamentul produsului, ceea ce clientul percepe drept calitatea lui, este dat în bună măsură de programul care rulează în acel microcontroler. Un reductor prost proiectat se aude; un algoritm de comandă prost proiectat se simte la fel de tare, doar că nimeni nu îl poate deschide cu șurubelnița pentru a-l vedea.

Este util să trecem prin câteva domenii în care microcontrolerul este piesa centrală, nu pentru a face un inventar, ci pentru a observa că cerințele tehnice diferă mult de la un domeniu la altul și că, tocmai de aceea, există sute de familii de microcontrolere în loc de una singură.

1.1.1 Robotică și mecatronică

Un sistem mecatronic este, prin definiție, un ansamblu în care partea mecanică, partea electronică și partea de program sunt proiectate împreună. Microcontrolerul este locul unde se închide bucla: el citește senzorii, calculează comanda și o trimite spre elementele de execuție.

Într-un braț robotic sau într-un dispozitiv de prehensare, microcontrolerul citește poziția fiecărei articulații de la un traductor incremental, compară poziția măsurată cu cea dorită și modifică tensiunea aplicată motorului, de regulă printr-un semnal modulat în durată. Această buclă trebuie parcursă integral de sute sau de mii de ori pe secundă, iar durata unei parcurgeri trebuie să fie constantă: un regulator proporțional-integrativ-derivativ presupune, în formulele lui, un pas de eșantionare fix. Dacă pasul variază de la o iterație la alta, termenul derivativ produce vârfuri de comandă care se aud și se văd în mișcarea brațului.

Controlul mișcării nu este singura sarcină. Un robot mobil are nevoie de *fuziunea senzorilor*, adică de combinarea informației de la mai multe surse care spun lucruri parțial contradictorii: un accelerometru, un giroscop, un senzor de distanță cu ultrasunete, câteva contacte de capăt de cursă. Fiecare dintre ele are propriul zgomot, propria întârziere și propria rată de actualizare, iar microcontrolerul trebuie să le citească pe toate, să le aducă la un numitor comun de timp și să extragă o estimare unică a stării robotului. Sarcina aceasta amestecă lucruri pe care le vom studia pe rând în capitolele următoare: conversia analog-numerică, întreruperile, temporizatoarele și protocoalele de comunicație serială.

1.1.2 Automobilul

Automobilul modern este probabil cel mai dens sistem încorporat produs în serie mare. Un autoturism obișnuit conține câteva zeci de unități electronice de comandă, legate între ele printr-o rețea de comunicație, iar fiecare unitate are în interior cel puțin un microcontroler.

Unitatea de comandă a motorului calculează, pentru fiecare ciclu al fiecărui cilindru,

momentul aprinderii și cantitatea de combustibil injectată, pornind de la turație, de la presiunea din galeria de admisie, de la temperatura lichidului de răcire și de la concentrația de oxigen din gazele de evacuare. La o turație mare, intervalul dintre două aprinderi succesive scade sub o milisecundă, iar comanda trebuie livrată cu o eroare de câteva grade de rotație a arborelui cotit. Aici nu este suficient ca microcontrolerul să fie rapid în medie; el trebuie să fie punctual de fiecare dată, ceea ce este o cerință complet diferită.

Sistemul antiblocare la frânare măsoară viteza fiecărei roți și, când constată că o roată se apropie de blocare, reduce presiunea din circuitul de frână al roții respective, apoi o restabilește, de mai multe ori pe secundă. Sistemele de climatizare, de diagnoză, de alarmă, de comandă a geamurilor și a oglinzilor sunt toate construite în jurul unor microcontrolere mai modeste, alese astfel încât costul lor să fie cât mai mic, pentru că se montează în milioane de exemplare.

1.1.3 Electronica de consum și aparatura casnică

În electronica de consum, microcontrolerul este de obicei creierul care gestionează interfața cu utilizatorul și coordonează circuitele specializate. Un sistem audio, un televizor, o cameră video, un receptor de navigație prin satelit sau o jucărie electronică au, fiecare, un microcontroler care citește tastele și telecomanda, aprinde afșajul, ține minte setările utilizatorului și comandă restul circuitelor.

Aparatura electrocasnică este exemplul clasic, pentru că acolo se vede cel mai bine ce a schimbat microcontrolerul. O mașină de spălat din anii 1970 avea un programator electromecanic: un motor sincron rotea un ax cu came, iar camele închideau și deschideau contacte, în ordinea și la momentele fixate de forma lor. Un program nou însemna came noi. Astăzi, aceeași funcție este îndeplinită de un microcontroler care măsoară temperatura apei, urmărește nivelul, comandă electrovalva, rezistența de încălzire și motorul tamburului, detectează dezechilibrul încărcăturii după curentul absorbit și afișează timpul rămas. Programul nou înseamnă acum o altă versiune de firmware, iar producătorul poate oferi zece cicluri de spălare în loc de trei fără să schimbe nimic în mecanică. Aceeași observație se aplică frigiderului, cuptorului cu microunde și aspiratorului.

1.1.4 Controlul mediului și instrumentația

În controlul mediului, microcontrolerul trebuie în primul rând să măsoare corect și să acționeze rar. Într-o seră, într-o locuință sau într-o hală industrială se urmăresc temperatura, umiditatea, concentrația de dioxid de carbon, iluminarea, iar în funcție de acestea se comandă ventilatoare, clapete, umbriri și instalații de încălzire. Constantele de timp ale acestor procese sunt de ordinul minutelor sau al orelor, deci puterea de calcul necesară este neînsemnată. În schimb, contează foarte mult stabilitatea măsurătorii pe termen lung, robustețea la perturbații electrice și, dacă echipamentul este alimentat de la baterie, consumul propriu.

În instrumentație, microcontrolerul a înlocuit atât partea de afișare, cât și o bună parte din prelucrarea analogică. Un aparat de măsură numeric conține un convertor analog-

numeric și un microcontroler care liniarizează caracteristica senzorului, aplică corecțiile de temperatură, calculează media pe un număr de eșantioane și afișează rezultatul. Aici a apărut și noțiunea de *senzor inteligent*: un traductor care are în aceeași capsulă elementul sensibil și un microcontroler mic, astfel încât la ieșire nu mai apare o tensiune care trebuie interpretată, ci o valoare deja calibrată, transmisă pe o magistrală digitală. Avantajul este că informația nu se mai degradează pe cablu și că fiecare exemplar își poartă cu sine propria calibrare.

Din toate aceste exemple se desprinde o concluzie care va organiza restul cărții: microcontrolerul nu este interesant prin puterea lui de calcul, care este modestă, ci prin faptul că este singurul component care leagă lumea fizică (tensiuni, curenți, timpi, mișcări) de lumea logică a unui program.

1.2 Ce este un microcontroler

După ce am văzut unde apare, putem da o definiție. Un *microcontroler* este un microcircuit care încorporează, în aceeași capsulă și pe același cip de siliciu, o unitate centrală de prelucrare, memorie și un set de resurse care îi permit să interacționeze direct cu mediul exterior. Cuvântul cheie al definiției este „încorporează”: nu este vorba de un procesor căruia i se adaugă memorie și periferice, ci de un circuit care le conține deja pe toate și care, alimentat și prevăzut cu o sursă de ceas, funcționează singur.

De aici vine și numele. Un microprocesor *procesează* date care îi sunt aduse din exterior; un microcontroler *controlează* un proces, ceea ce presupune că are pini prin care măsoară și comandă. Un microcontroler căruia i-am lua perifericele nu ar mai avea niciun rost, pentru că nu ar mai avea cu ce să atingă lumea.

1.2.1 Resursele minime

Pentru ca un circuit să merite numele de microcontroler, el trebuie să conțină un set minim de blocuri. În primul rând o unitate centrală de prelucrare, adică unitatea aritmetică și logică, registrele de lucru și logica de decodificare a instrucțiunilor, împreună cu un oscilator care să genereze semnalul de ceas, de obicei cu posibilitatea de a funcționa fără niciun component extern.

În al doilea rând, memorie locală de două feluri. O memorie nevolatilă, în care stă programul și care își păstrează conținutul când dispare alimentarea: de-a lungul timpului aceasta a fost de tip ROM, PROM, EPROM, iar astăzi este aproape întotdeauna de tip Flash. Și o memorie volatilă de tip RAM, în care stau variabilele, stiva și zonele de lucru și al cărei conținut se pierde la întreruperea alimentării.

În al treilea rând, un sistem de întreruperi, adică un mecanism prin care un eveniment exterior sau interior poate suspenda execuția programului curent și poate impune execuția unei rutine dedicate. Fără întreruperi, un microcontroler ar fi obligat să interogheze permanent toate sursele de evenimente, ceea ce ar consuma tot timpul procesorului și ar face imposibil un răspuns rapid.

În al patrulea rând, linii de intrare și de ieșire numerice, grupate în porturi paralele, prin

care microcontrolerul citește contacte, comută tranzistoare și dialoghează cu alte circuite. Și, în sfârșit, un sistem de temporizatoare și numărătoare programabile, care sunt singurul mod prin care un program poate măsura timpul cu precizie, fără să irosească procesorul în bucle de așteptare.

1.2.2 Resursele opționale

Peste acest nucleu, fiecare familie adaugă periferice care o fac potrivită pentru un anumit tip de aplicație. Cele mai des întâlnite sunt următoarele:

- un sistem de conversie analog-numerică, prin care tensiunile provenite de la senzori devin numere;
- porturi seriale sincrone și asincrone programabile, pentru dialogul cu alte circuite sau cu un calculator;
- conversie numeric-analogică sau, mult mai frecvent, ieșiri cu modulare în durată, prin care se comandă motoare, elemente de încălzire sau surse de lumină;
- un comparator analogic, util pentru detecția rapidă a depășirii unui prag, fără a mai trece prin conversie;
- o memorie de date nevolatilă de tip EEPROM, în care se păstrează parametrii de calibrare și setările utilizatorului între două alimentări;
- facilități suplimentare pentru temporizatoare, în special captarea (memorarea automată a momentului în care apare un front pe un pin) și compararea (generarea automată a unui eveniment la un moment programat);
- circuite de supraveghere a funcționării, dintre care cel mai important este temporizatorul de gardă, care repornește sistemul dacă programul încetează să îl mai reinițializeze;
- mecanisme de reducere a consumului propriu, adică regimuri de repaus în care ceasul este oprit total sau parțial.

Această listă nu trebuie memorată acum. Este harta perifericelor pe care le vom studia, pe rând, în capitolele următoare, iar rostul ei aici este să arate că diferența dintre două microcontrolere nu stă atât în procesor, cât în ce anume poartă acesta după el.

1.3 Microprocesor, microcontroler și sistem pe cip

Cele trei denumiri sunt adesea folosite una în locul alteia, dar ele desemnează trei moduri diferite de a împărți un sistem de calcul în circuite integrate. Distincția nu este academică: ea decide câte componente are placa, cât costă produsul, cât consumă și cât de repede răspunde.

Un *microprocesor* conține practic numai unitatea centrală: unitatea aritmetică și logică, registrele, unitatea de comandă și interfața cu magistralele externe. Memoria și perifericele sunt circuite separate, legate de procesor prin magistrala de adrese, magistrala de date și liniile de comandă. Un sistem construit în jurul unui microprocesor are deci pe placă, în afară de procesor, cel puțin un circuit de memorie nevolatilă, unul sau mai multe circuite de memorie RAM, circuite de interfață și logica de decodificare a adreselor. Placa este mare, are multe conexiuni rapide care trebuie rutate cu grijă, iar consumul este pe măsură. În

schimb, memoria poate fi extinsă practic oricât, iar procesorul poate fi mult mai puternic decât nucleul oricărui microcontroler.

Un *microcontroler* aduce pe același siliciu procesorul, memoria de program, memoria de date și perifericele. Consecințele sunt directe. Sistemul minim se reduce, în cazuri fericite, la circuit, un condensator de decuplare și alimentare. Magistralele interne nu mai ies din capsulă, deci nu mai radiază și nu mai sunt sensibile la perturbații. Accesul la periferice se face în câteva cicluri de ceas, nu printr-un ciclu de magistrală externă. În schimb, resursele sunt fixate la fabricație: dacă programul nu încapă în memoria Flash a circuitului, nu se adaugă un al doilea circuit de memorie, ci se schimbă microcontrolerul.

Un *sistem pe cip* (în engleză *system on chip*, prescurtat SoC) duce ideea mai departe, dar într-o altă direcție. Un SoC integrează unul sau mai multe nuclee de procesor de mare putere, un procesor grafic, controlere de memorie externă, interfețe rapide și blocuri specializate de prelucrare a semnalului. Diferența esențială față de microcontroler este că un SoC nu își poartă, de regulă, memoria principală în interior: el are un controler de memorie și folosește circuite externe de tip DRAM. Un SoC rulează aproape întotdeauna un sistem de operare complet, are nevoie de mai multe tensiuni de alimentare bine secvențiate și de o placă cu mai multe straturi. Telefonul mobil și televizorul inteligent sunt construite în jurul unor SoC-uri, dar tot ele conțin, pe lângă SoC, mai multe microcontrolere mici care se ocupă de baterie, de tastatură sau de gestiunea alimentării, tocmai pentru că acelea pornesc instantaneu și consumă puțin.

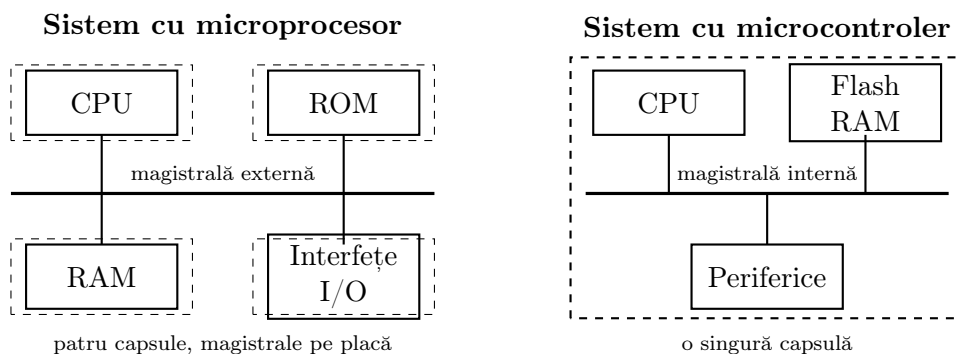


Figura 1.1 Aceeași funcție, două partiționări. La stânga, procesorul, memoriile și interfețele sunt circuite distincte, legate printr-o magistrală care iese pe placă. La dreapta, toate blocurile se află în interiorul aceleiași capsule.

Figura 1.1 rezumă diferența. Trebuie subliniat că granițele nu sunt perfect nete: există microcontrolere care pot adresa memorie externă și există SoC-uri mici care au și memorie internă suficientă pentru a funcționa fără DRAM. Tabelul 1.1 adună criteriile după care se face, în practică, alegerea.

Merită insistat asupra unei rubrici din tabel, cea privitoare la program. Într-un calculator de birou, programele se schimbă permanent: utilizatorul instalează, șterge, actualizează. Într-un microcontroler, programul este de obicei scris o singură dată, la fabricarea produsului, și rămâne același toată viața acestuia. De aici decurg consecințe pe care le vom întâlni la

Tabela 1.1 Deosebirile practice dintre microprocesor, microcontroler și sistem pe cip.

Criteriu	Microprocesor	Microcontroler	Sistem pe cip
Memorie	externă, extensibilă	internă, fixată la fabricație	controler intern, DRAM externă
Periferice	circuite separate	pe același siliciu	pe același siliciu, dar complexe
Program	încărcat de pe disc, se schimbă des	fix, scris o dată în Flash	sistem de operare complet
Resurse tipice	memorie mare, frecvență mare	kiloocteți de memorie, zeci de MHz	memorie mare, sute de MHz
Consum	mare	foarte mic	mediu spre mare
Cost pe bucată	mediu spre mare	foarte mic	mare
Pornire	lentă	imediată	lentă

tot pasul: nu există alocare dinamică a memoriei în sensul obișnuit, pentru că nu există cine să facă ordine după ce memoria se fragmentează; nu există sistem de fișiere; nu există un sistem de operare care să separe programele între ele, pentru că nu există decât un singur program. Programul nostru este singurul stăpân al mașinii și, tocmai de aceea, este și singurul responsabil de tot ce se întâmplă.

Observație

Un microcontroler nu „pornește” în sensul în care pornește un calculator. Din momentul în care tensiunea de alimentare a atins pragul, după o scurtă temporizare de stabilizare, nucleul începe să execute instrucțiunea de la adresa zero. Nu există sistem de operare care să se încarce, nu există dispozitiv de pornire de ales. De aceea un aparat construit în jurul unui microcontroler este gata de lucru în câteva milisecunde, iar unul construit în jurul unui SoC are nevoie de zeci de secunde.

1.4 Sistemul încorporat și criteriile lui de proiectare

Un microcontroler nu se folosește niciodată singur. El este miezul unui *sistem încorporat*, adică al unui ansamblu în care partea de calcul servește o funcție bine definită, fixată la proiectare, într-un produs care nu se prezintă drept calculator. Structura acestui ansamblu este aproape întotdeauna aceeași: senzorii transformă mărimile fizice în semnale electrice, un lanț de condiționare le aduce în domeniul acceptat de microcontroler, microcontrolerul le convertește în numere, calculează comanda și o trimite prin elementele de execuție înapoi către procesul fizic, care își modifică starea, care este apoi din nou măsurată. Figura 1.2 arată această buclă.

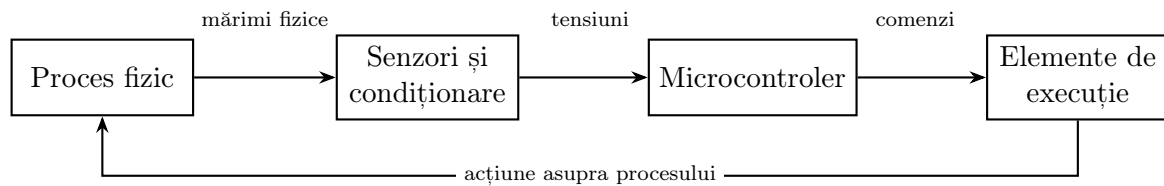


Figura 1.2 Bucla unui sistem încorporat. Timpul de parcurgere a buclei, de la apariția unei modificări în proces până la efectul comenzii, este mărimea care decide dacă sistemul funcționează sau nu.

Odată desenată această buclă, devine limpede de ce criteriile după care se alege și se programează un microcontroler nu seamănă cu cele după care se alege un calculator de birou. Le luăm pe rând.

1.4.1 Determinismul

La un calculator de birou, singura întrebare este cât durează o operație în medie. Dacă o pagină se deschide în 0,2 secunde de obicei și o dată la o sută de ori în 2 secunde, nimeni nu se plânge serios. Într-un sistem încorporat, întrebarea relevantă este cu totul alta: care este durata cea mai mare posibilă? Un sistem este *determinist* dacă putem indica, pentru fiecare reacție a lui, o limită superioară a întârzierii, garantată în orice condiții.

Diferența nu este de nuanță. Un sistem de aprindere care lucrează corect de un milion de ori și greșește o dată produce o rată, iar dacă acest lucru se întâmplă la fiecare câteva minute, motorul funcționează neregulat. Un declanșator de airbag care răspunde de obicei în cinci milisecunde, dar uneori în cincizeci, este inutilizabil, pentru că cele cincizeci de milisecunde suplimentare se consumă în timp ce ocupantul se deplasează spre volan.

Determinismul se pierde ușor și din motive care par nevinovate. O funcție de bibliotecă a cărei durată depinde de valoarea datelor, o buclă de așteptare a unui periferic care uneori nu răspunde, o rutină de întrerupere lungă care întârzie o altă mai urgentă, o memorie cache care uneori conține instrucțiunea căutată și alteori nu: toate acestea introduc variabilitate. Un microcontroler mic are avantajul, aparent paradoxal, că este simplu: fără cache, fără predicție de salt, fără sistem de operare care să comute contexte, durata unei secvențe de instrucțiuni poate fi calculată exact, numărând ciclii de ceas din foaia de catalog.

Exemplu

Nucleul ATmega328P execută majoritatea instrucțiunilor într-un singur ciclu de ceas. La frecvența de lucru de 16 MHz a plăcii Arduino Uno, aceasta înseamnă cam 16 milioane de instrucțiuni pe secundă. Dacă bucla de reglare a unui motor trebuie parcursă la fiecare milisecundă, câte instrucțiuni avem la dispoziție într-o iterație?

Rezolvare. O milisecundă înseamnă $16\,000\,000 \times 0,001 = 16\,000$ cicli de ceas, deci aproximativ 16 000 de instrucțiuni simple. Pare mult, dar trebuie scăzut ce se consumă în afara calculului propriu-zis: o conversie analog-numerică pe acest circuit durează în jur de 100 de microsecunde, adică deja o zecime din perioadă, iar transmiterea unui caracter

pe interfața serială la 9600 de biți pe secundă durează aproximativ o milisecundă, adică o perioadă întreagă. Concluzia practică, la care vom reveni de mai multe ori, este că depășirea termenului nu vine din calcule, ci din așteptarea perifericelor. De aceea vom învăța să nu așteptăm niciodată un periferic într-o buclă, ci să lucrăm cu întreruperi.

1.4.2 Timpul de răspuns

Strâns legat de determinism, dar distinct de el, este timpul de răspuns, adică intervalul dintre apariția unui eveniment fizic și momentul în care sistemul a produs efectul cerut. El se compune din întârzierea senzorului, din întârzierea de detecție a evenimentului de către microcontroler, din durata calculului și din inerția elementului de execuție.

Un sistem se numește *în timp real* nu pentru că este rapid, ci pentru că un răspuns corect livrat prea târziu este considerat greșit. Se face deosebirea între timp real strict, unde depășirea termenului înseamnă avarie sau accident, și timp real relaxat, unde depășirea ocazională degradează calitatea, dar nu compromite funcția. Comanda injectiei este timp real strict; actualizarea afișajului cu temperatura este timp real relaxat. Aceeași aplicație conține de obicei sarcini din ambele categorii, iar arta proiectării constă în a le da prioritățile potrivite.

1.4.3 Consumul

Un calculator de birou este alimentat de la rețea și disipă zeci sau sute de wați. Un sistem încorporat este adesea alimentat de la o baterie care trebuie să țină luni sau ani, iar în acest caz consumul devine criteriul dominant. Un senzor fără fir care raportează o temperatură din zece în zece minute petrece peste 99% din timp în repaus; energia consumată în acele momente de inactivitate, înmulțită cu durata lor, depășește cu mult energia consumată în timpul măsurării.

Mecanismul fizic este util de reținut, pentru că explică toate tehnicile de economisire pe care le vom folosi. Într-un circuit digital realizat în tehnologie CMOS, o poartă consumă energie în principal atunci când comută, pentru că la fiecare comutare se încarcă sau se descarcă o capacitate parazită. Puterea dinamică este deci proporțională cu frecvența de comutare și cu pătratul tensiunii de alimentare. De aici rezultă imediat trei pârgarii: coborârea frecvenței de ceas, coborârea tensiunii de alimentare și oprirea ceasului acolo unde nu este nevoie de el. Ultima este cea mai eficientă, pentru că un bloc al cărui ceas este oprit nu comută deloc, deci nu consumă putere dinamică; îi rămâne doar curentul de scurgere, mult mai mic. Toate regimurile de repaus ale microcontrolerelor sunt, în fond, combinații de opriri selective de ceas.

1.4.4 Costul unitar

Într-un produs fabricat în serie mare, costul microcontrolerului contează la bănuț. Diferența de un leu între două circuite înseamnă un milion de lei la un milion de bucăți, sumă din care se poate plăti o echipă de proiectanți un an întreg. De aceea, în sistemele încorporate, se

alege în mod curent circuitul cel mai mic care face treaba, chiar dacă asta înseamnă efort de programare mai mare, în timp ce la un calculator de birou se cumpără mai multă memorie decât este nevoie, tocmai ca să nu se piardă timp cu optimizări.

Costul nu se reduce însă la prețul circuitului. Un microcontroler cu mai multe periferice integrate poate fi mai scump decât altul, dar dacă elimină de pe placă un convertor analog-numeric extern, un circuit de ceas de timp real și un stabilizator, costul total al produsului scade. Tot la cost se adaugă numărul de straturi al cablajului, numărul de componente pasive și, nu în ultimul rând, timpul necesar testării fiecărei bucăți pe linia de fabricație.

1.4.5 Fiabilitatea

Un calculator de birou care se blochează este repornit de utilizator. Un microcontroler dintr-un contor de energie montat pe stâlp, dintr-un implant medical sau dintr-o unitate de comandă a motorului nu are lângă el pe cineva care să apese un buton. Sistemul trebuie deci să fie capabil să își detecteze singur starea anormală și să iasă din ea.

Instrumentul clasic este *temporizatorul de gardă*, cunoscut și sub numele englezesc *watchdog*: un numărător independent, care merge pe ceasul lui și care, dacă ajunge la capăt, generează un reset al microcontrolerului. Programul corect îl reinițializează periodic, la momente bine alese din bucla principală. Dacă programul s-a blocat într-o buclă infinită sau a ajuns, din cauza unei perturbații, să execute cod fără sens, reinițializarea nu mai are loc, temporizatorul ajunge la capăt și sistemul repornește. Trebuie subliniat că temporizatorul de gardă nu repară nimic: el doar limitează durata unei defecțiuni. Folosit greșit, adică reinițializat dintr-o rutină de întrerupere periodică ce merge indiferent de starea programului principal, își pierde complet rostul, pentru că va continua să fie hrănit și după ce programul principal a murit.

La fiabilitate contribuie și lucruri mai simple: comportarea corectă la scăderea tensiunii de alimentare, pentru că un microcontroler alimentat sub tensiunea minimă poate executa instrucțiuni greșite și poate corupe memoria EEPROM; robustețea la perturbații electromagnetice, motiv pentru care condensatorul de decuplare de lângă pinii de alimentare nu este o formalitate; și verificarea integrității datelor păstrate în memoria nevolatilă.

Punând cap la cap aceste cinci criterii, se vede de ce proiectarea unui sistem încorporat este un exercițiu de economie. Un calculator de birou se proiectează adăugând resurse până când performanța este mulțumitoare; un sistem încorporat se proiectează scoțând resurse până când sistemul abia mai reușește să își respecte termenele, pentru că fiecare resursă rămasă costă bani, curent și o posibilă defecțiune în plus.

1.5 Scurt istoric al microcontrolerelor

Istoria microcontrolerului începe cu istoria microprocesorului, iar aceasta începe cu o comandă comercială obișnuită. La sfârșitul anilor 1960, compania japoneză Busicom, producătoare de calculatoare de birou, avea nevoie de un set de circuite specializate pentru o nouă gamă de produse și s-a adresat unor firme americane. Soluția convențională ar fi fost proiectarea câte unui circuit dedicat pentru fiecare model, ceea ce însemna un efort de

proiectare care se irosea la fiecare produs nou.

Ideea care a schimbat lucrurile, atribuită lui Marcian Hoff, inginer de origine română la Intel, a fost să nu se mai proiecteze un circuit pentru fiecare funcție, ci un singur circuit capabil să execute o secvență de instrucțiuni memorate. Funcția produsului urma să fie dată de conținutul memoriei de program, nu de topologia siliciului. Astfel, în 1971, Intel a lansat 4004, un procesor cu o arhitectură pe patru biți, cu un semnal de ceas sub un megahertz, capabil de câteva mii de operații pe secundă și însoțit de circuite de memorie și de intrare-ieșire care completeau setul. Anul următor a apărut 8008, pe opt biți, mai capabil, iar de aici linia s-a ramificat rapid.

Pasul de la microprocesor la microcontroler s-a făcut la mijlocul anilor 1970, când a devenit posibil ca pe același cip să încapă, alături de procesor, și memoria de program, și memoria de date, și liniile de intrare-ieșire. Texas Instruments a produs în 1974 familia TMS1000, gândită de la început ca un calculator complet într-o singură capsulă pentru aparate de larg consum. Intel a răspuns în 1976 cu familia 8048, iar în 1980 cu 8051, arhitectură care a avut o carieră extraordinară de lungă: nuclee compatibile cu 8051 se mai produc și astăzi, la peste patru decenii de la lansare. Tot din a doua jumătate a anilor 1970 vine și linia PIC, pornită la General Instrument și dezvoltată mai târziu de Microchip.

Anii 1980 sunt deceniul consacării microcontrolerelor de opt biți. În această perioadă apar, integrate pe cip, perifericele care ne interesează în această carte: temporizatoarele programabile, interfețele de comunicație serială și primele convertoare analog-numerice. Cu ele, microcontrolerul a devenit suficient pentru aplicații casnice și de control industrial, iar cuptorul cu microunde și mașina de spălat au căpătat programator electronic.

Anii 1990 aduc arhitecturile pe 16 și pe 32 de biți în zona sistemelor încorporate. Creșterea lungimii cuvântului nu înseamnă doar numere mai mari; ea înseamnă spațiu de adresare mult mai larg, deci programe mai mari, și calcule mai comode cu mărimi care nu încap în opt biți. În aceeași perioadă, la mijlocul anilor 1990, Atmel introduce arhitectura AVR, proiectată de la început pentru a fi programată eficient în limbajul C, cu un set de instrucțiuni regulat și cu 32 de registre generale. Din aceeași familie face parte și microcontrolerul pe care îl vom studia. Tot atunci se generalizează memoria Flash în locul memoriei EPROM ștersă cu ultraviolete, ceea ce schimbă radical modul de lucru al proiectantului: reprogramarea devine o operație de câteva secunde, care se poate face fără a scoate circuitul din montaj.

Anii 2000 aduc integrarea perifericelor de comunicație complexe, USB în primul rând, și rafinarea celor existente: temporizatoare cu moduri specializate pentru comanda motoarelor, convertoare cu rezoluție și viteză mai bune, controlere de comunicație pentru automobil. Tot în acest deceniu apare arhitectura ARM Cortex-M, gândită anume pentru microcontrolere, cu un tratament al întreruperilor proiectat pentru latență mică și previzibilă. Nucleele Cortex-M sunt licențiate de zeci de producători, ceea ce a avut o consecință importantă: același nucleu, deci același compilator și aceleași deprinderi de programare, se regăsește în circuite foarte diferite ca periferice și ca preț.

În jurul anului 2010, microcontrolerele pe 32 de biți devin dominante în aplicațiile noi de

complexitate medie, memoria integrată crește considerabil, iar consumul scade prin tehnologii de fabricație mai fine și prin regimuri de repaus mai bine gândite. După 2015, răspândirea aplicațiilor de tip Internet al lucrurilor a împins microcontrolerele către integrarea interfețelor radio, îndeosebi Wi-Fi și comunicații de arie largă cu consum redus, astfel încât un singur circuit poate măsura, decide și raporta. Direcțiile actuale continuă această linie prin doi vectori: arhitectura deschisă RISC-V, apărută dintr-un proiect universitar de la începutul anilor 2010 și adoptată pentru că nu implică plata unei licențe pentru setul de instrucțiuni, și execuția pe microcontroler a unor modele reduse de învățare automată, pentru recunoașterea de cuvinte-cheie sau detecția de mișcare direct pe dispozitiv.

Observație

Ar fi greșit să se înțeleagă din acest istoric că microcontrolerele de opt biți au fost înlocuite. Ele se produc și se vând în continuare în cantități uriase, pentru că într-o telecomandă, într-o periută electrică sau într-un termostat un nucleu pe 32 de biți nu aduce niciun folos, dar aduce cost și consum în plus. Evoluția a adăugat categorii noi în vârf, nu le-a șters pe cele de la bază.

1.6 Anatomia unui microcontroler

Trecem acum de la ce face un microcontroler la cum este el construit. Ne interesează nivelul de bloc, adică felul în care circuitul arată din punctul de vedere al programatorului, nu al proiectantului de siliciu. Figura 1.3 prezintă structura, iar în cele ce urmează parcurgem blocurile unul câte unul.

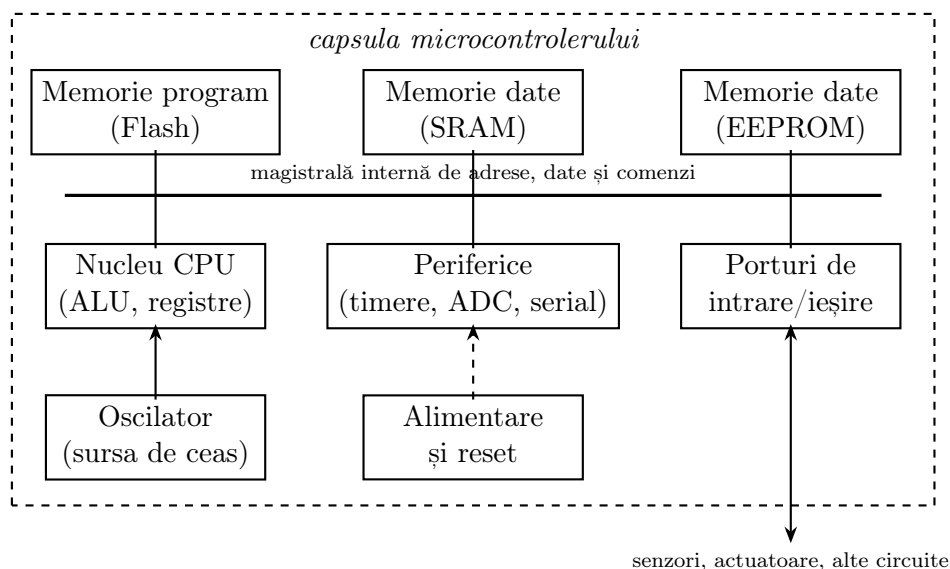


Figura 1.3 Schema-bloc a unui microcontroler. Toate blocurile din interiorul conturului punctat se află pe același cip; în afară rămân doar sursa de alimentare, eventualul cristal de cuarț și lumea pe care circuitul o măsoară și o comandă.

1.6.1 Nucleul

Nucleul este partea care execută instrucțiunile. El conține unitatea aritmetică și logică, unde se fac adunările, comparațiile și operațiile pe biți, registrele de lucru, în care stau operanzii imediat înainte și imediat după operație, contorul de program, care indică adresa instrucțiunii următoare, indicatorul de stivă și registrul de stare, ale cărui fanioane rețin dacă rezultatul ultimei operații a fost zero, negativ sau a produs transport.

Nucleele de microcontroler sunt, de regulă, de tip RISC: setul de instrucțiuni este mic, instrucțiunile au lungime uniformă și majoritatea se execută într-un singur ciclu de ceas. Alegerea nu este întâmplătoare. Un set de instrucțiuni regulat înseamnă o unitate de comandă mică, deci puțin siliciu, deci cost mic; iar execuția într-un ciclu înseamnă că durata unei secvențe se poate calcula exact, ceea ce am văzut că este esențial pentru determinism.

O deosebire importantă privește organizarea memoriei. În arhitectura *von Neumann*, programul și datele stau în același spațiu de adrese și se citesc pe aceeași magistrală, ceea ce înseamnă că nucleul nu poate aduce în același timp o instrucțiune și un operand. În arhitectura *Harvard*, memoria de program și memoria de date sunt separate fizic, cu magistrale proprii, astfel încât o instrucțiune poate fi adusă în timp ce se citește sau se scrie un operand al instrucțiunii precedente. Majoritatea microcontrolerelor mici, între care și cel pe care îl vom folosi, sunt de tip Harvard, tocmai pentru a obține un debit de o instrucțiune pe ciclu fără a complica nucleul. Prețul plătit este că datele constante păstrate în memoria de program nu se citesc cu aceleași instrucțiuni ca variabilele din memoria de date, chestiune la care vom reveni când va fi vorba de economisirea memoriei SRAM.

1.6.2 Memoria de program

Memoria de program păstrează codul executabil și constantele. Ea este nevolatilă, adică își păstrează conținutul și după deconectarea alimentării, pentru că altfel produsul ar trebui reprogramat la fiecare pornire. Istoric, această memorie a fost mai întâi de tip ROM, scrisă o singură dată chiar la fabricarea cipului, apoi PROM, programabilă o singură dată de utilizator, apoi EPROM, ștersă prin expunere la lumină ultravioletă printr-o fereastră de cuarț practică în capsulă, apoi EEPROM, ștersă electric. Astăzi este aproape întotdeauna memorie *Flash*, care se șterge electric, pe blocuri, și se scrie de la câteva mii la câteva zeci de mii de ori.

Numărul limitat de cicluri de ștergere este o proprietate fizică, nu o convenție: fiecare celulă memorează informația sub forma unei sarcini electrice prinse pe o poartă flotantă izolată, iar ștergerea, care presupune trecerea purtătorilor prin stratul izolator, degradează treptat acest strat. După un număr suficient de cicluri, celula nu mai reține sarcina destul de mult timp. Pentru memoria de program limita nu deranjează, pentru că reprogramarea se face de câteva zeci sau sute de ori în timpul dezvoltării și niciodată după aceea. Devine însă o problemă reală pentru memoria de date nevolatilă, dacă programul scrie în ea prea des.

Dimensiunea memoriei de program este primul criteriu de alegere a microcontrolerului

și este fixată definitiv la cumpărare. Un program care nu încapă trebuie fie rescris mai economic, fie mutat pe un alt circuit.

1.6.3 Memoria de date

Memoria de date de lucru este de tip SRAM: rapidă, scrisă și citită de un număr nelimitat de ori, dar volatilă. În ea stau variabilele globale, zona de lucru a funcțiilor, stiva și eventualele tampoane de comunicație. Este resursa cea mai des subestimată de începători, pentru că este, în mod tipic, de zece sau de douăzeci de ori mai mică decât memoria de program.

Depășirea memoriei SRAM nu se manifestă la compilare, ci în timpul funcționării, și în modul cel mai neplăcut cu putință. Stiva crește de la o extremitate a zonei de memorie către cealaltă, în timp ce variabilele ocupă zona de la capătul opus. Când adâncimea de apel a funcțiilor crește suficient, stiva ajunge peste variabile și le suprascrie. Programul continuă să ruleze, dar cu valori schimbate, iar la revenirea dintr-o funcție poate ajunge, prin adresa de revenire coruptă, oriunde în memoria de program. Simptomul este un sistem care merge corect ore întregi și apoi se repornește fără motiv aparent, de obicei exact atunci când este arătat clientului. De aici regulile pe care le vom respecta: fără alocare dinamică, tampoane dimensionate explicit, atenție la variabilele locale mari și la funcțiile care se apelează în cascadă.

Multe microcontrolere au și o a treia memorie, de tip EEPROM, nevolatilă, dar scrisă cuvânt cu cuvânt și destinată datelor care trebuie să supraviețuiască opririi: constante de calibrare, ore de funcționare, ultima setare aleasă de utilizator. Este mult mai lentă la scriere decât SRAM, iar numărul de scrieri pe celulă este limitat, deci nu se folosește pentru date care se schimbă des.

1.6.4 Perifericele

Perifericele sunt blocuri de circuite specializate care lucrează independent de nucleu și care comunică cu el prin registre. Aceasta este ideea centrală a întregii cărți și merită formulată apăsător: un periferic nu este o funcție de bibliotecă, ci un circuit real, care funcționează și atunci când procesorul face altceva. Programul nu îi cere să numere impulsuri; îi scrie într-un registru configurația dorită, iar din acel moment perifericul numără singur, pe ceasul lui, până când programul se întoarce să îi citească rezultatul sau până când perifericul însuși cere atenție printr-o întrerupere.

Consecința practică este că a programa un microcontroler înseamnă, în cea mai mare parte, a scrie și a citi registre. Fiecare periferic are un mic set de registre, fiecare registru are biți cu semnificație precisă, iar foaia de catalog a circuitului este documentul care le descrie. Vom folosi deci constant scrieri de forma următoare, în care nu se atribuie o valoare întregului registru, ci se modifică un singur bit, lăsându-i pe ceilalți neatinși.

Exemplu 1.1 Comanda unui bit de port prin registre: se configurează linia ca ieșire, apoi se pune în 1.

```

1 DDRB |= (1 << PB5);    /* bitul 5 al portului B devine iesire */
2 PORTB |= (1 << PB5);    /* linia este pusa in 1 logic      */
3 PORTB &= ~(1 << PB5);    /* linia este pusa in 0 logic      */

```

Din categoria perifericelor fac parte porturile de intrare-ieșire numerice, temporizatoarele, convertorul analog-numeric, interfețele seriale, generatoarele de semnal modulat în durată, comparatorul analogic și blocul de întreruperi externe. Fiecăruia dintre acestea îi este dedicat, mai departe, cel puțin un capitol.

1.6.5 Oscilatorul

Nimic nu se mișcă într-un circuit digital sincron fără semnalul de ceas. *Oscilatorul* produce acest semnal, iar frecvența lui determină direct viteza de execuție, rezoluția de măsurare a timpului și, așa cum am văzut, consumul.

Un microcontroler are de obicei mai multe surse de ceas la dispoziție, iar alegerea între ele este un compromis. Un oscilator RC integrat pe cip nu cere niciun component extern și pornește repede, dar frecvența lui variază cu temperatura și cu tensiunea de alimentare, tipic cu câteva procente. Este perfect pentru a aprinde un afișaj sau a citi un buton și complet nepotrivit pentru o comunicație serială asincronă, unde emițătorul și receptorul trebuie să fie de acord asupra duratei unui bit cu o eroare de cel mult câteva procente cumulate pe un caracter. Un rezonator ceramic extern este mai bun, iar un cristal de cuarț dă o precizie cu câteva ordine de mărime mai bună, la prețul a două componente suplimentare, al unui timp de pornire mai lung și al unui consum ceva mai mare. Aceasta este și rațiunea pentru care placa Arduino Uno are un cristal montat pe ea.

1.6.6 Alimentarea, resetul și pinii

Ultimul bloc este cel care leagă circuitul de lumea reală. Alimentarea se aplică pe perechea de pini de tensiune pozitivă și masă, iar în imediata lor vecinătate trebuie așezat un condensator de decuplare. Rostul lui este fizic: la fiecare front de ceas, mii de porți comută simultan și cer un vârf de curent pe care firele de alimentare, cu inductanța lor, nu îl pot livra instantaneu. Condensatorul, aflat la câțiva milimetri, joacă rolul unui rezervor local. Fără el, tensiunea de alimentare are căderi scurte la fiecare comutare, iar circuitul funcționează imprevizibil, cu erori care apar rar și sunt foarte greu de diagnosticat.

Circuitul de reset aduce microcontrolerul într-o stare inițială cunoscută. El acționează la aplicarea alimentării, la comanda externă a pinului de reset, la scăderea tensiunii sub un prag configurabil și la expirarea temporizatorului de gardă. După reset, toate registrele au valorile prevăzute în foaia de catalog, iar execuția începe de la adresa zero.

Restul pinilor sunt liniile de intrare-ieșire. Aproape întotdeauna ele sunt multiplexate: același pin fizic poate fi linie numerică obișnuită, intrare a convertorului analog-numeric, ieșire a unui temporizator sau linie a unei interfețe seriale, în funcție de configurația scrisă în registre. Multiplexarea este consecința directă a economiei: pinii sunt scumpi, pentru că fiecare cere suprafață de siliciu pentru circuitele de protecție și de comandă și mărește

capsula. Ea are însă și o consecință practică asupra proiectării: două funcții care au nevoie de același pin nu pot fi folosite simultan, iar acest lucru trebuie verificat în tabelul de multiplexare înainte de a desena cablajul, nu după.

Observație

Pentru un microcontroler, foaia de catalog nu este documentație opțională, ci unealta principală de lucru. Ea conține harta registrelor, semnificația fiecărui bit, duratele exacte ale operațiilor, limitele electrice ale pinilor și schemele de conectare recomandate. Un programator de microcontrolere care nu deschide foaia de catalog lucrează prin încercări, iar rezultatul funcționează, cel mult, pe exemplarul de pe masă.

1.7 Microcontrolerul acestui curs: ATmega328P și platforma Arduino

Pentru a învăța ceva trebuie ales un exemplu concret, iar alegerea unui exemplu înseamnă întotdeauna renunțarea la altele. În această carte lucrăm cu ATmega328P, un microcontroler de opt biți din familia AVR, montat pe placa Arduino Uno. Alegerea nu este făcută pentru că acest circuit ar fi cel mai performant, pentru că nu este, ci pentru că este cel mai potrivit pentru învățat.

Motivul principal este că ATmega328P este suficient de simplu pentru a fi înțeles în întregime. Foaia lui de catalog are câteva sute de pagini, ceea ce pare mult, dar este de zece ori mai puțin decât manualul de referință al unui microcontroler modern pe 32 de biți. Nu are memorie cache, nu are unitate de gestiune a memoriei, nu are magistrale interne multiple cu arbitraj, nu are un arbore de ceasuri cu zeci de divizoare care trebuie configurate înainte ca vreun periferic să dea semne de viață. Un student poate parcurge, în timpul unui semestru, drumul complet de la un bit dintr-un registru până la o aplicație care funcționează, și poate să înțeleagă fiecare pas al acestui drum. deprinderile astfel formate se transferă apoi, aproape neschimbate, la orice alt microcontroler: registrele au alte nume, dar mecanismul este același.

Al doilea motiv este mediul. Platforma Arduino oferă o placă ieftină și robustă, alimentată prin cablul de programare, cu un încărcător rezident în memoria Flash care permite programarea fără niciun aparat suplimentar, și un mediu de dezvoltare care se instalează în câteva minute. Bariera de intrare este practic nulă, iar timpul câștigat astfel se investește în înțelegerea circuitului. Există în plus posibilitatea simulării, utilă atunci când placa nu este la îndemână.

Tabelul 1.2 adună cifrele. Merită citite cu atenție, pentru că ele fixează cadrul în care vom lucra tot restul cărții și pentru că ordinul lor de mărime este surprinzător pentru cineva obișnuit cu calculatoarele actuale.

Cei 32 de kiloocteți de memorie Flash reprezintă tot spațiul disponibil pentru program și pentru constante. Este suficient pentru aplicații serioase, dar nu suficient pentru risipă: o singură bibliotecă folosită fără socoteală poate ocupa câteva mii de octeți. Din acest spațiu, o mică parte, situată la capătul superior al memoriei, este ocupată de încărcătorul rezident, care primește programul nou pe interfața serială și îl scrie în Flash.

Tabela 1.2 Principalele resurse ale microcontrolerului ATmega328P, în configurația de pe placa Arduino Uno.

Resursă	Valoare
Arhitectură	AVR, RISC, Harvard, cuvânt de date de 8 biți
Frecvență de lucru	16 MHz pe placa Arduino Uno, dată de un cristal de cuarț
Memorie de program	32 KB Flash, din care o parte este ocupată de încărcător
Memorie de date	2 KB SRAM
Memorie nevolatilă de date	1 KB EEPROM
Registre generale	32, de câte 8 biți, accesibile direct de unitatea aritmetică
Linii de intrare-ieșire	23, grupate în trei porturi
Temporizatoare	două de 8 biți și unul de 16 biți, cu moduri de captare, comparare și modulare în durată
Conversie analog-numerică	convertor cu aproximații succesive, rezoluție de 10 biți, cu multiplexor de intrare
Interfețe seriale	una asincronă de tip USART, una sincronă de tip SPI, una de tip două fire
Alimentare	5 V pe placa Arduino Uno

Cei doi kiloocteți de SRAM sunt însă adevărata constrângere. Tot ce înseamnă variabile, tampoane și stivă trebuie să încapă în 2048 de octeți.

Exemplu

Un program trebuie să păstreze ultimele 100 de valori măsurate, pentru a calcula o medie alunecătoare, iar programatorul le declară de tipul `float`. Cât din memoria SRAM ocupă acest tampon și ce se poate face?

Rezolvare. Pe această arhitectură, un `float` ocupă 4 octeți, deci tamponul cere $100 \times 4 = 400$ de octeți, adică aproape 20% din cei 2048 de octeți disponibili. Dacă valorile provin de la convertorul analog-numeric, ele au însă doar 10 biți, deci încap într-un `uint16_t`, adică în 2 octeți. Cu această schimbare tamponul scade la 200 de octeți, iar suma valorilor, care poate ajunge la $100 \times 1023 = 102\,300$, se păstrează într-un `uint32_t`. Se câștigă astfel 200 de octeți și, în plus, calculul devine mai rapid, pentru că nucleul nu are unitate de calcul în virgulă mobilă, iar operațiile cu `float` sunt realizate prin rutine de bibliotecă, de zeci de ori mai lente decât operațiile cu întregi.

Exemplul de mai sus arată tipul de raționament care va reveni permanent: alegerea tipului de date nu mai este o chestiune de comoditate, ci una de resurse.

Ultimul lucru care merită spus despre platforma Arduino privește raportul dintre funcțiile

ei și registrele circuitului. Mediul Arduino oferă funcții simple, precum cele care configurează și comandă un pin, iar acestea sunt utile pentru a verifica repede o idee. Ele nu sunt însă magie: fiecare dintre ele se traduce, în cele din urmă, în scrieri în aceleași registre pe care le-am amintit mai sus, plus un cost în timp de execuție și în memorie, cost pe care programatorul trebuie să îl cunoască. Următorul fragment face același lucru în ambele stiluri.

Exemplu 1.2 Aceeasi actiune, exprimata prin functiile Arduino si direct prin registre.

```

1  /* varianta cu functii Arduino */
2  void setup(void) {
3      pinMode(13, OUTPUT);
4  }
5  void loop(void) {
6      digitalWrite(13, HIGH);
7      delay(500);
8      digitalWrite(13, LOW);
9      delay(500);
10 }
11
12 /* aceeasi actiune, direct pe registre */
13 void setup(void) {
14     DDRB |= (1 << PB5);           /* pinul 13 este PB5 */
15 }
16 void loop(void) {
17     PORTB |= (1 << PB5);
18     _delay_ms(500);
19     PORTB &= ~(1 << PB5);
20     _delay_ms(500);
21 }

```

Cele două variante aprind și sting același led. Varianta cu funcții este mai lizibilă pentru un începător; varianta cu registre este mai scurtă, mai rapidă și, mai important pentru noi, arată exact ce se întâmplă în circuit. În această carte vom folosi ambele, dar explicația va porni întotdeauna de la registre, pentru că numai acolo se vede mecanismul.

1.8 Harta cărții

Restul volumului urmează o ordine care nu este întâmplătoare: fiecare capitol folosește ce s-a construit în cele dinainte, iar la capăt toate se întâlnesc într-o aplicație completă.

Capitolul următor se ocupă de arhitectura internă a microcontrolerului AVR: cum sunt organizate memoriile, ce fac cele 32 de registre generale, cum se execută o instrucțiune și cum se citesc, în foaia de catalog, informațiile de care avem nevoie. Este capitolul care transformă schema-bloc din Figura 1.3 într-o hartă concretă de adrese.

Urmează programarea microcontrolerului: lanțul de compilare, structura unui program

pentru un sistem fără sistem de operare, particularitățile limbajului C în acest context și modul în care programul ajunge din calculator în memoria Flash a circuitului.

Capitolul despre porturile de intrare-ieșire este primul contact real cu lumea exterioară. Acolo se studiază structura electrică a unui pin, cele trei registre care îl guvernează, rezistențele de ridicare la tensiunea de alimentare, curenții admiși și problema contactelor mecanice, care nu comută niciodată curat.

Sistemul de întreruperi este capitolul care schimbă modul de gândire. Până acolo, programul face un singur lucru pe rând; de acolo încolo, evenimentele externe pot interveni oricând, iar programul trebuie scris astfel încât să rămână corect în aceste condiții. Se studiază vectorii de întrerupere, latența, prioritățile, regulile de scriere a unei rutine de tratare și problema variabilelor partajate.

Modulele de tip temporizator arată cum se măsoară și cum se generează timpul fără a ocupa procesorul: numărare, prescalare, comparare, captare și generarea semnalelor modulate în durată, cu care se comandă motoarele și se reglează luminozitatea.

Convertorul analog-numeric leagă mărimile continue de programul nostru. Se parcurge lanțul complet, de la traductor și circuitul de condiționare până la valoarea numerică, cu discuția erorilor: eșantionare, cuantizare, referință de tensiune, zgomot.

Două capitole sunt dedicate comunicațiilor. Primul tratează transmisia serială asincronă, cu formatul cadrului, generarea vitezei de transmisie și erorile posibile. Al doilea se ocupă de magistralele sincrone, cu arbitraj și adresare, prin care microcontrolerul dialoghează cu senzori și cu circuite periferice.

Ultimele două capitole schimbă registrul. Unul este dedicat metodei de proiectare și dezvoltare a unei aplicații cu microcontroler, de la cerințe la punerea în funcțiune, cu accent pe depanare și pe verificare. Celălalt duce totul într-o aplicație unitară, un robot mobil, în care porturile, întreruperile, temporizatoarele, convertorul și comunicația serială lucrează împreună și în care se vede, în sfârșit, de ce fiecare dintre ele a fost necesar.

Recomandarea de studiu este simplă și fermă: fiecare capitol trebuie citit cu foaia de catalog alături și cu placa pe masă. Un microcontroler nu se învață citind despre el, la fel cum nu se învață înotul citind despre apă. Explicațiile din carte lămuresc mecanismul; certitudinea că l-ați înțeles vine abia atunci când montajul face ce ați prevăzut că va face.

Întrebări de verificare

1. Explicați, pornind de la modul în care este partiționat sistemul în circuite integrate, de ce un sistem construit în jurul unui microprocesor are pe placă mai multe capsule decât unul construit în jurul unui microcontroler și ce consecințe are aceasta asupra costului și a sensibilității la perturbații.
2. Un producător de electrocasnice trebuie să aleagă între un microcontroler de opt biți și un sistem pe cip pentru comanda unui cuptor cu microunde. Ce argumente aduceți în favoarea fiecărei soluții și care credeți că va decide?
3. Ce înseamnă că un sistem este determinist și de ce absența memoriei cache și a predicției

- de salt, care într-un calculator de birou ar fi un dezavantaj, devine aici un avantaj?
4. Distingeți între timp real strict și timp real relaxat, dând câte un exemplu din automobil pentru fiecare și explicând ce se întâmplă concret la depășirea termenului.
 5. Puterea dinamică disipată de un circuit CMOS depinde de frecvența de comutare și de pătratul tensiunii de alimentare. Deduceți din aceasta trei metode de reducere a consumului și explicați de ce oprirea ceasului unui bloc nefolosit este mai eficientă decât reducerea frecvenței de lucru.
 6. Explicați rolul temporizatorului de gardă și arătați de ce reinițializarea lui dintr-o rutină de întrerupere periodică îi anulează utilitatea.
 7. Care este deosebirea dintre arhitectura von Neumann și arhitectura Harvard și ce avantaj concret aduce a doua într-un microcontroler care urmărește să execute o instrucțiune pe ciclu de ceas?
 8. Descrieți mecanismul prin care depășirea memoriei SRAM duce la un comportament aparent aleatoriu al programului și indicați trei măsuri de proiectare prin care acest lucru poate fi evitat.
 9. De ce memoria Flash și memoria EEPROM suportă un număr limitat de cicluri de scriere, în timp ce memoria SRAM nu are această limitare? Ce consecință are acest lucru asupra unui program care ar vrea să înregistreze în EEPROM o valoare la fiecare secundă?
 10. Un program trebuie să păstreze 250 de eșantioane provenite de la convertorul analog-numeric de 10 biți al circuitului ATmega328P. Calculați memoria necesară dacă se folosește tipul `float` și dacă se folosește un tip întreg potrivit, și exprimați rezultatul ca procent din cei 2 KB de SRAM disponibili.
 11. Considerând că nucleul execută majoritatea instrucțiunilor într-un ciclu de ceas la 16 MHz, estimați câte instrucțiuni pot fi executate într-o perioadă de eșantionare de 200 de microsecunde și discutați dacă în acest interval încapă o conversie analog-numerică completă.
 12. Explicați de ce multiplexarea funcțiilor pe același pin este inevitabilă la un microcontroler și ce trebuie verificat, înainte de a desena cablajul, atunci când o aplicație are nevoie simultan de mai multe periferice.

2. Arhitectura unui microcontroler

- Bitul, octetul și citirea unei valori
- Registrii
- Registrii cu funcții speciale
- Memoria unui microcontroler
- Unitatea centrală de procesare
- Semnalele la interfața unității centrale cu exteriorul
- Arhitectura generală a unui microcontroler
- Setul de instrucțiuni
- Arhitectura ATmega328P
- Circuitul oscilator
- Ciclul de instrucțiune
- Alimentarea microcontrolerului
- Un exemplu de conectare
- Întrebări de verificare

În capitolul precedent am privit microcontrolerul din afară: un circuit integrat cu câteva zeci de picioare, care citește senzori și comandă actuatoare. Acum îl deschidem. Scopul acestui capitol este să răspundă la o întrebare care pare simplă, dar care nu este: ce se întâmplă, fizic și logic, între momentul în care scriem o linie de cod și momentul în care un LED se aprinde.

Răspunsul trece prin patru straturi. Primul este reprezentarea informației, adică modul în care o mărime din lumea reală devine un șir de zerouri și de unu. Al doilea este memoria, locul în care acele șiruri stau și așteaptă. Al treilea este unitatea centrală de procesare, mașina care le citește, le transformă și le scrie înapoi. Al patrulea este ceasul, semnalul periodic care dă tuturor acestor operații ritmul lor. Un inginer care înțelege aceste patru straturi poate citi orice foaie de catalog și poate depana orice montaj; unul care nu le înțelege va rămâne prizonierul funcțiilor gata făcute din biblioteci și se va bloca la prima problemă care nu seamănă cu exemplele.

Microcontrolerul de referință rămâne ATmega328P, circuitul de pe placa Arduino Uno. Vom folosi cifrele lui concrete ori de câte ori este posibil, pentru că un manual care vorbește numai despre „un microcontroler generic” lasă cititorul fără puncte de sprijin.

2.1 Bitul, octetul și citirea unei valori

Un microcontroller nu știe ce este un număr, o literă sau o temperatură. Tot ce știe el este că într-un anumit punct al circuitului tensiunea este apropiată de tensiunea de alimentare sau este apropiată de masă. Această alegere între două stări este singura informație elementară pe care hardware-ul o poate păstra fără ambiguitate, și ea poartă numele de *bit*, prescurtare de la *binary digit*, cifră binară. Convențional, tensiunea apropiată de alimentare se notează cu 1 (nivel logic înalt, *high*), iar cea apropiată de masă cu 0 (nivel logic coborât, *low*).

Motivul pentru care electronica digitală a ales două stări, și nu zece, este de robustețe. Dacă am vrea să codificăm zece nivele într-un domeniu de 5 V, fiecare nivel ar ocupa o jumătate de volt, iar orice zgomot de comutație, orice cădere de tensiune pe un fir lung, orice variație cu temperatura ar putea transforma un 6 într-un 7. Cu două stări, un circuit care primește 0,4 V știe sigur că este vorba de zero, iar unul care primește 4,2 V știe sigur că este vorba de unu; între ele există o zonă largă, interzisă, care absoarbe zgomotul. Pentru un dispozitiv care lucrează într-un automobil, lângă un motor, sau într-o hală industrială, această marjă nu este un lux.

Un singur bit nu spune însă mare lucru. Pentru a reprezenta ceva util, biții se grupează. Grupul fundamental este de opt biți și se numește *octet* (în engleză *byte*). Alegerea lui opt are rădăcini istorice, dar s-a dovedit fericită: un octet poate lua $2^8 = 256$ valori distincte, ceea ce este suficient pentru a acoperi alfabetul latin cu semne de punctuație și cifre, așa cum face codificarea ASCII, și este exact rezoluția uzuală a unui convertor analog-numeric simplu sau a unui canal de modulație în lățime de impuls.

Octetul este și unitatea în care se măsoară dimensiunea memoriilor. Multiplii lui se formează în puteri ale lui doi, nu ale lui zece, pentru că adresele de memorie sunt ele însele numere binare: un kilooctet (KB) înseamnă 1024 de octeți, un megaoctet (MB) înseamnă 1024 KB, un gigaoctet (GB) înseamnă 1024 MB, iar un teraoctet (TB) înseamnă 1024 GB. Când citim că ATmega328P are 2 KB de memorie SRAM, înțelegem deci 2048 de octeți, nu 2000.

2.1.1 Aceeași valoare, trei scrieri

Un octet este un obiect fizic: opt bistabile, fiecare într-o stare sau în cealaltă. Modul în care îl scriem pe hârtie este însă o convenție, iar în practică folosim trei convenții în paralel, fiecare potrivită unei situații.

Baza doi sau scrierea binară transcrie direct starea fiecărui bistabil. Ea este singura scriere în care se vede pinul: dacă bitul 5 este 1, atunci pe pinul corespunzător lui vom găsi tensiune. Dezavantajul ei este lungimea, opt caractere pentru un singur octet, și faptul că ochiul obosește repede numărând poziții.

Baza zece sau scrierea zecimală este limbajul nostru natural și singurul în care putem judeca mărimi. Când vrem să știm dacă valoarea citită de la un senzor este mare sau mică, o citim în zecimal. Dezavantajul ei este că nu are nicio legătură vizibilă cu biții: din numărul 180 nu se vede care pin este activ.

Baza șaisprezece sau scrierea hexazecimală este compromisul. Deoarece $16 = 2^4$, fiecare cifră hexazecimală acoperă exact patru biți, adică o jumătate de octet, numită și *nibble*. Un octet se scrie deci întotdeauna cu exact două cifre hexazecimale, iar trecerea între hexazecimal și binar se face pe grupe de patru, mecanic, fără calcul. Din acest motiv toate foile de catalog dau adresele și valorile implicate ale regiștrilor în hexazecimal.

Cifrele hexazecimale sunt de la 0 la 9, apoi A, B, C, D, E, F pentru valorile de la 10 la 15. În limbajul C, o constantă binară se scrie cu prefixul `0b`, una hexazecimală cu prefixul `0x`, iar una zecimală fără prefix.

Exemplu

Un registru de configurare a fost citit și conține valoarea binară 1011 0100. Să se exprime această valoare în zecimal și în hexazecimal și să se spună care biți sunt activi.

Rezolvare. Numerotăm biții de la dreapta la stânga, începând cu 0. Șirul 1011 0100 are deci bitul 7 egal cu 1, bitul 6 egal cu 0, bitul 5 egal cu 1, bitul 4 egal cu 1, bitul 3 egal cu 0, bitul 2 egal cu 1, bitul 1 egal cu 0 și bitul 0 egal cu 0.

Valoarea zecimală se obține adunând ponderile biților egali cu 1, unde ponderea bitului de rang n este 2^n :

$$2^7 + 2^5 + 2^4 + 2^2 = 128 + 32 + 16 + 4 = 180.$$

Valoarea hexazecimală se obține împărțind șirul în două grupe de câte patru biți: 1011 și 0100. Prima grupă valorează $8 + 2 + 1 = 11$, adică cifra B; a doua valorează 4, adică cifra 4. Rezultă `0xB4`. Verificarea este imediată: $11 \cdot 16 + 4 = 180$.

Biții activi sunt deci 7, 5, 4 și 2. Dacă acest octet ar fi fost scris într-un registru de direcție a unui port, cei patru pini corespunzători ar fi devenit ieșiri, iar ceilalți patru ar fi rămas intrări.

Numerotarea biților nu este arbitrară. Bitul 0 este cel mai puțin semnificativ, *least significant bit* sau LSB, pentru că schimbarea lui modifică valoarea cu o unitate. Bitul 7 este cel mai semnificativ, *most significant bit* sau MSB, pentru că schimbarea lui modifică valoarea cu 128. În toate foile de catalog, bitul 7 se desenează în stânga și bitul 0 în dreapta, exact ca la scrierea unui număr zecimal obișnuit, unde cifra cea mai grea stă în stânga. Figura 2.1 arată această convenție pentru valoarea din exemplul de mai sus.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
1	0	1	1	0	1	0	0
MSB				LSB			

Figura 2.1 Structura unui registru pe 8 biți. Conținutul desenat corespunde valorii `0b10110100 = 180 = 0xB4`.

Această numerotare este cea care face ca lucrul pe biți să fie limbajul natural al micro-

controlerului. Un procesor de calculator personal manipulează structuri de date complexe și lasă hardware-ul în seama sistemului de operare; un microcontroler manipulează pini. Iar un pin nu are decât două stări, deci un bit. Ori de câte ori vrem să aprindem un LED, să citim un buton, să pornim un numărător sau să activăm o întrerupere, operația de fond este aceeași: schimbăm un singur bit dintr-un octet aflat la o adresă cunoscută. Restul, funcțiile din biblioteci, tipurile de date, obiectele, sunt straturi așezate peste această operație elementară.

2.2 Regiștrii

Un *registru* este o celulă de memorie de foarte mare viteză, integrată chiar în interiorul unității centrale sau al unui modul periferic, la care procesorul are acces imediat. Diferența față de memoria propriu-zisă nu este de natură, ci de amplasare și de cost: un registru este construit din bistabile așezate în imediata vecinătate a unității aritmetice, astfel încât citirea sau scrierea lui să nu ceară un ciclu de magistrală. Tocmai pentru că sunt scumpi în suprafață de siliciu, regiștrii sunt puțini, în timp ce memoria este ieftină și abundentă.

Rolul lor este de a păstra temporar datele și adresele cu care unitatea centrală lucrează chiar acum. O operație aritmetică nu se poate face direct între două locații de memorie: valorile trebuie mai întâi aduse în regiștri, prelucrate acolo, iar rezultatul dus înapoi în memorie. Această asimetrie explică de ce un program scris neglijent, care mută mereu date între memorie și regiștri, este mult mai lent decât unul care își ține variabilele de lucru în regiștri cât mai mult timp; compilatorul de C face această optimizare automat, dar numai dacă îi lăsăm loc.

După funcția pe care o îndeplinesc, regiștrii se împart în câteva familii. Regiștrii *de date* păstrează operanzii și rezultatele intermediare ale calculelor. Regiștrii *de adresă* păstrează adrese de memorie, adică locul de unde urmează să se citească sau unde urmează să se scrie; pe AVR, perechile de regiștri X, Y și Z joacă acest rol și permit parcurgerea unui tablou cu incrementare automată. Regiștrii *de control* sau de configurare păstrează setări care determină comportamentul unui modul: dacă un numărător este pornit sau oprit, cu ce factor de divizare lucrează, dacă un pin este intrare sau ieșire. În sfârșit, regiștrii *de stare* raportează condiții: dacă ultima operație aritmetică a dat zero, dacă a produs transport, dacă o conversie analog-numerică s-a terminat.

Distincția între ultimele două familii este esențială pentru practică și merită formulată explicit, pentru că este sursa unei confuzii frecvente la începători. Un registru de control este scris de program și citit de hardware: noi punem acolo o comandă, iar circuitul o execută. Un registru de stare este scris de hardware și citit de program: circuitul raportează acolo ce s-a întâmplat, iar noi luăm cunoștință. A încerca să pornim un convertor scriind în registrul lui de stare este la fel de zadarnic ca a încerca să pornim un motor mișcând acul turometrului.

2.3 Regiștrii cu funcții speciale

Regiștrii despre care am vorbit până acum deservește unitatea centrală. Există însă o a doua categorie, mult mai interesantă pentru cel care construiește aplicații: *regiștrii cu funcții*

speciale, în literatura de limbă engleză *special function registers* sau SFR. Ei nu sunt locuri de depozitare, ci interfețe. Fiecare astfel de registru este legat prin fire fizice la un circuit intern concret: un numărător, convertorul analog-numeric, modulul de comunicație serială, blocul care comandă pinii, oscilatorul.

Consecința este remarcabilă și trebuie înțeleasă bine, pentru că pe ea se sprijină tot restul cărții. Regiștrii cu funcții speciale sunt *mânerele* prin care programul comandă perifericele. Nu există un alt canal, nu există comenzi secrete: tot ce poate face un microcontroler se face scriind sau citind biți la anumite adrese. Scriem un bit într-o adresă și, în aceeași fracțiune de microsecundă, se schimbă ceva fizic: un pin trece de la zero la cinci volți, un numărător începe să numere, un convertor pornește o conversie. Invers, citim un bit de la o adresă și aflăm ceva despre lumea din afară: dacă un buton este apăsător, dacă a sosit un caracter pe linia serială.

În interiorul unui astfel de registru, fiecare bit are propria lui semnificație și propriul lui nume, dat în foaia de catalog. Biții *de control* operează direct asupra hardware-ului: setarea lor pornește sau oprește o funcție. Biții *de stare* sunt operați direct de hardware: ei marchează îndeplinirea unei condiții, cum ar fi rezultat nul, transport sau depășire. Regula practică pe care merită să o reținem este simplă: în general scriem biții de control și citim biții de stare.

Cel mai la îndemână exemplu este comanda pinilor. Pe ATmega328P, pinii sunt grupați în porturi notate cu litere, iar fiecare port este deservit de trei regiștri. Registrul *DDRx (data direction register)* stabilește direcția: un bit pus pe 1 face pinul corespunzător ieșire, un bit pus pe 0 îl lasă intrare. Registrul *PORTx* stabilește, pentru un pin configurat ca ieșire, nivelul care se aplică la exterior; pentru un pin configurat ca intrare, același registru activează sau dezactivează rezistența internă de ridicare la alimentare. Registrul *PINx* returnează, la citire, starea electrică reală măsurată pe pini.

Exemplu 2.1 Comanda unui pin prin regiștrii cu funcții speciale

```

1 // LED-ul de pe placa Arduino Uno este legat la pinul PB5.
2 // Butonul este legat la PD2, cu rezistența internă de ridicare.
3
4 DDRB |= (1 << PB5); // PB5 devine ieșire: pun pe 1 bitul 5 din
    ↪ DDRB
5 DDRD &= ~(1 << PD2); // PD2 devine intrare: pun pe 0 bitul 2
    ↪ din DDRD
6 PORTD |= (1 << PD2); // activez rezistența internă de ridicare
    ↪ pe PD2
7
8 PORTB |= (1 << PB5); // aprind LED-ul
9 PORTB &= ~(1 << PB5); // sting LED-ul
10 PORTB ^= (1 << PB5); // inversez starea LED-ului
11

```

```

12 // Citirea starii butonului: apasat inseamna nivel logic 0,
13 // pentru ca butonul leaga pinul la masa.
14 if ((PIND & (1 << PD2)) == 0) {
15     PORTB |= (1 << PB5);
16 }

```

Merită descompus ce se întâmplă în aceste linii, pentru că sintaxa se repetă în toată programarea de microcontrolere. Expresia $(1 \ll PB5)$ construiește o *mască*: numărul 1, care în binar are numai bitul 0 activ, este deplasat la stânga cu cinci poziții, rezultând 0b00100000. Operatorul $|=$ efectuează un SAU logic bit cu bit între conținutul actual al registrului și mască: bitul 5 devine 1, iar ceilalți rămân neatinși, pentru că un SAU cu 0 nu schimbă nimic. Operatorul $\&=$ împreună cu mască negată prin $\sim$ efectuează un ȘI logic cu 0b11011111: bitul 5 devine 0, ceilalți rămân neatinși, pentru că un ȘI cu 1 nu schimbă nimic. Operatorul $\^=$ efectuează un SAU-EXCLUSIV, care inversează exact bitul acoperit de mască.

Observație

Tentația de a scrie `PORTB = 0b00100000`; în loc de `PORTB |= (1 << PB5)`; este mare, pentru că pare mai clar. Diferența este că prima formă scrie toți cei opt biți, stingând tot ce era aprins pe ceilalți șapte pini ai portului. Într-un program mic, unde portul B servește un singur LED, greșeala nu se vede. Într-un program în care portul B mai comandă un releu și o linie de selecție SPI, efectul apare abia după săptămâni, sub forma unui defect intermitent imposibil de reprodus. Regula este deci: se modifică bitul dorit, nu se rescrie registrul, cu excepția inițializărilor în care chiar dorim să impunem o stare completă.

2.4 Memoria unui microcontroler

Memoria stochează informațiile necesare funcționării corecte a microcontrolerului. Ea se împarte, după destinație, în două mari categorii. *Memoria de program* conține instrucțiunile, adică traducerea în cod mașină a programului scris de noi; ea trebuie să reziste la întreruperea alimentării, altfel dispozitivul ar fi gol la fiecare pornire. *Memoria de date* conține variabilele, rezultatele intermediare și stiva; ea se modifică permanent în timpul execuției și, în cea mai mare parte, poate fi volatilă.

Indiferent de tip, memoria este organizată ca un șir de locații, fiecare cu o *adresă* unică, la fel ca numerele de pe o stradă. Unitatea centrală nu caută niciodată o valoare, ci se duce direct la adresa ei. După drepturile de acces, locațiile pot fi numai de citire (R), numai de scriere (W) sau de citire și scriere; în practică, într-un microcontroler coexistă mai multe tipuri de memorie cu proprietăți diferite, iar alegerea celei potrivite pentru fiecare dată este o decizie de proiectare.

2.4.1 Memorii volatile

Memoria *RAM* (*random access memory*, memorie cu acces aleator) este folosită pentru stocarea temporară a datelor. Numele ei subliniază proprietatea esențială: orice locație este accesibilă în orice ordine, cu aceeași durată de acces, spre deosebire de o bandă magnetică sau de o memorie serială, unde a ajunge la sfârșit costă mai mult decât a ajunge la început. Memoria RAM poate fi citită și scrisă de unitatea centrală, dar își pierde conținutul la întreruperea alimentării.

Memoria *DRAM* (*dynamic RAM*) folosește o celulă formată dintr-un condensator și un tranzistor. Sarcina din condensator se scurge, deci conținutul trebuie înprospătat periodic, ceea ce cere un controler dedicat. În schimb celula este foarte mică, deci memoria este ieftină pe bit; acesta este tipul folosit pentru memoria principală a calculatoarelor.

Memoria *SRAM* (*static RAM*) folosește o celulă construită din patru până la șase tranzistoare, montate ca două inversoare în buclă. Bucla se menține singură cât timp există alimentare, deci nu are nevoie de înprospătare și lucrează la viteze mari. Prețul plătit este suprafața: o celulă SRAM ocupă de câteva ori mai mult siliciu decât una DRAM. Din acest motiv SRAM se folosește acolo unde viteza contează și cantitatea este mică, adică la memoria cache a procesoarelor și la memoria de date a microcontrolerelor.

2.4.2 Memorii nevolatile

Memoria *ROM* (*read only memory*) păstrează date permanente pe care utilizatorul nu le poate modifica, sau le poate modifica numai cu mare dificultate: programe de pornire, firmware, constante care trebuie să fie disponibile imediat după alimentare. Ea poate fi doar citită.

Memoria *PROM* (*programmable ROM*) este nevolatilă și poate fi programată o singură dată de utilizator, prin distrugerea controlată a unor punți interne. Costul redus și robustețea au făcut-o utilă în producția de serie mică. Memoria *EPROM* (*erasable PROM*) poate fi ștersă prin expunere la radiație ultravioletă, printr-o fereastră de cuarț practică în capsulă, și reprogramată; este robustă, dar ștergerea cere scoaterea circuitului din montaj și zeci de minute de iradiere. Memoria *MaskedROM* își primește conținutul chiar în procesul de fabricație, printr-o mască fotolitografică; este cea mai ieftină soluție la volume foarte mari, dar conținutul nu mai poate fi schimbat niciodată.

Memoria *Flash* este nevolatilă și reprogramabilă electric, fără scoaterea din montaj. Particularitatea ei este că ștergerea nu se poate face pe locații individuale, ci pe blocuri întregi, numite pagini; de aceea se spune că ștergerea este neselectivă. În schimb suportă un număr mare de cicluri de ștergere și scriere, de ordinul a zece mii pentru memoriile de program ale microcontrolerelor uzuale. Majoritatea microcontrolerelor actuale folosesc memorie Flash pentru stocarea programului, tocmai pentru că permite reprogramarea în circuit, deci actualizarea firmware-ului la un produs deja livrat.

Memoria *EEPROM* (*electrically erasable PROM*) este tot nevolatilă și reprogramabilă electric, dar permite ștergerea selectivă, la nivel de octet. Este folosită pentru datele de

configurație care trebuie să supraviețuiască întreruperii alimentării. Are două limitări importante: numărul de cicluri de scriere și ștergere este limitat, de ordinul a o sută de mii pe locație, iar viteza de scriere este mică, de ordinul milisecundelor pentru un octet. Aceste două limitări au aceeași consecință practică: EEPROM-ul nu se scrie într-o buclă. Un program care salvează o valoare în EEPROM la fiecare trecere prin bucla principală va epuiza celula în câteva ore de funcționare.

Tabela 2.1 Comparație între principalele tipuri de memorie folosite în sistemele încorporate.

Tip	Volatilă	Rescriere	Utilizare tipică
SRAM	da	nelimitată	variabile, stivă, memorie cache
DRAM	da	nelimitată	memoria principală a calculatoarelor
MaskedROM	nu	imposibilă	firmware fixat la fabricație
PROM	nu	o singură dată	serii mici, date de identificare
EPROM	nu	cu ultraviolete	prototipuri, sisteme mai vechi
Flash	nu	pe pagini	memoria de program a microcontrolerelor
EEPROM	nu	pe octet	parametri de configurație, calibrări

2.5 Unitatea centrală de procesare

Unitatea centrală de procesare (UCP, în engleză *central processing unit* sau CPU) monitorizează și controlează toate procesele din interiorul microcontrolerului. Ea este partea care aduce instrucțiunile din memorie, le interpretează și le execută, generând semnalele care pun în mișcare restul circuitului. Structural, unitatea centrală se împarte în două blocuri funcționale: unitatea aritmetică și logică, respectiv unitatea de control.

2.5.1 Unitatea aritmetică și logică

Unitatea aritmetică și logică (UAL, în engleză *arithmetic logic unit* sau ALU) este partea care calculează efectiv. Ea primește la intrare unul sau doi operanzi și un cod de operație, iar la ieșire produce un rezultat și un set de indicatori. Operațiile pe care le poate face sunt puține și elementare: adunare, scădere, comparație, ȘI, SAU, SAU-EXCLUSIV, negare, deplasare la stânga sau la dreapta, incrementare, decrementare. Tot ce înseamnă calcul complicat, de la înmulțire cu virgulă mobilă până la extragerea unei rădăcini, se construiește din aceste operații elementare, executate în secvență, prin program.

Operanzii nu vin din memorie, ci din regiștri. Arhitecturile clasice foloseau un registru privilegiat, numit *acumulator*, în care se aduna rezultatul fiecărei operații și care participa

obligatoriu la orice calcul. Arhitecturile moderne, între care și AVR, au renunțat la acumulatorul unic în favoarea unui *fișier de regiștri* generali, oricare dintre ei putând juca rolul de operand sau de destinație. ATmega328P are 32 de asemenea regiștri generali, notați de la R0 la R31, fiecare de opt biți. Prezența lor este motivul principal pentru care un AVR execută cod compilat din C mult mai eficient decât microcontrolerele mai vechi cu acumulator unic: compilatorul poate păstra simultan mai multe variabile în regiștri, fără să le mute înainte și înapoi în memorie.

Ultimii șase regiștri generali au și o funcție suplimentară. Perechile R27:R26, R29:R28 și R31:R30 formează regiștrii de adresă pe 16 biți X, Y și Z, folosiți pentru adresarea indirectă a memoriei de date; registrul Z poate adresa, în plus, memoria de program, ceea ce permite citirea unor tabele de constante păstrate în Flash.

2.5.2 Registrul de stare

Rezultatul unei operații aritmetice nu se rezumă la valoarea numerică. La fiecare operație, unitatea aritmetică și logică actualizează un *registru de stare*, notat pe AVR cu SREG, ai cărui biți se numesc *indicatori* sau *fanioane* (în engleză *flags*). Fiecare indicator răspunde la o întrebare despre operația tocmai încheiată.

Indicatorul Z (*zero*) se pune pe 1 dacă rezultatul a fost nul. Indicatorul C (*carry*) semnalează transportul ieșit din bitul cel mai semnificativ, adică faptul că rezultatul unei adunări fără semn nu a încăput în opt biți. Indicatorul N (*negative*) copiază bitul cel mai semnificativ al rezultatului, interpretat ca semn. Indicatorul V semnalează depășirea în aritmetica cu semn, care este o condiție diferită de transport. Indicatorul S este semnul corectat, obținut ca SAU-EXCLUSIV între N și V. Indicatorul H semnalează transportul dintre nibble-uri, util în aritmetica zecimală codificată binar. Indicatorul T este un bit de uz general, care poate primi sau furniza un bit oarecare dintr-un registru. În sfârșit, indicatorul I este cel care validează global întreruperile; fără el pe 1, nicio întrerupere nu este luată în seamă, oricât de bine ar fi configurate modulele.

Rostul acestor indicatori este să facă posibilă decizia. O instrucțiune de salt condiționat nu se uită la valori, ci la indicatori: instrucțiunea de comparație scade în gol cei doi operanzi și lasă indicatorii să spună dacă rezultatul a fost nul, negativ sau cu transport, iar saltul care urmează consultă indicatorul potrivit. Un `if` scris în C se traduce, aproape întotdeauna, într-o pereche formată dintr-o comparație și un salt condiționat.

Observație

Indicatorii se modifică la aproape fiecare instrucțiune aritmetică sau logică. Dacă între o comparație și saltul care depinde de ea se strecoară o altă operație, rezultatul saltului devine imprevizibil. În C acest lucru nu ne preocupă, pentru că se ocupă compilatorul; în limbaj de asamblare este una dintre cele mai frecvente surse de erori, alături de uitarea salvării registrului SREG la intrarea într-o rutină de tratare a întreruperilor.

2.5.3 Unitatea de control

Unitatea de control este partea care organizează execuția. Ea cuprinde numărătorul de program, blocul de instrucțiuni și blocul de adresă.

Numărătorul de program (PC, *program counter*) conține întotdeauna adresa următoarei instrucțiuni de executat. După fiecare aducere de instrucțiune el se incrementează automat, astfel încât programul avansează secvențial. Instrucțiunile de salt, de apel de subrutină și de întoarcere funcționează toate scriind o valoare nouă în numărătorul de program; nu există alt mecanism de ramificare.

Blocul de instrucțiuni realizează preluarea instrucțiunii din memoria de program, la adresa indicată de numărătorul de program, și decodificarea ei, adică determinarea operației care trebuie efectuată și a operanzilor implicați. Tot el generează semnalele de control care coordonează activitatea celorlalte componente: unitatea aritmetică și logică, regiștrii, unitatea de memorie.

Blocul de adresă calculează adresele necesare accesului la memorie. El interpretează *modurile de adresare* specificate de instrucțiuni: adresarea directă, în care adresa este scrisă în instrucțiune; adresarea indirectă, în care adresa se află într-un registru; adresarea relativă, în care se adună o deplasare la o valoare de bază; adresarea indexată și altele. ATmega328P dispune de cinci moduri diferite de adresare a memoriei de date, ceea ce permite parcurgerea eficientă a tablourilor și structurilor.

2.5.4 Stiva

Există o structură de date pe care hardware-ul o gestionează singur și pe care merită să o cunoaștem, pentru că este cauza unor defecte greu de explicat: *stiva* (*stack*). Ea este o zonă din memoria de date în care se depun temporar valori, după regula ultimul intrat, primul ieșit. Un registru special, *indicatorul de stivă* (*stack pointer*, pe AVR perechea de regiștri SPH:SPL), arată vârful stivei.

Stiva este folosită automat la apelul unei funcții: adresa de întoarcere, adică valoarea curentă a numărătorului de program, este depusă pe stivă înainte de saltul în funcție și este extrasă la revenire. Tot pe stivă se depun variabilele locale care nu încap în regiștri și contextul salvat la intrarea într-o rutină de tratare a întreruperilor.

Pe AVR, stiva începe la adresa cea mai mare a memoriei SRAM și crește în jos, spre adrese mici, în timp ce variabilele globale ocupă adresele mici și cresc în sus. Cele două zone înaintază una spre cealaltă. Dacă programul are prea multe variabile, sau dacă funcțiile se apelează pe prea multe nivele, sau dacă o funcție recursivă nu se oprește, cele două zone se suprapun: stiva scrie peste variabile sau variabilele scriu peste adresele de întoarcere. Rezultatul este un microcontroller care se comportă haotic, se resetează singur sau execută cod de la o adresă absurdă. Fenomenul se numește *depășire de stivă* și este, pe microcontrolerele cu memorie mică, unul dintre defectele cel mai greu de diagnosticat, tocmai pentru că simptomul apare departe de cauză.

2.6 Semnalele la interfața unității centrale cu exteriorul

Unitatea centrală comunică cu memoria și cu perifericele prin mănunchiuri de conductoare numite *magistrale* (în engleză *bus*). Funcțional, semnalele de la interfața unității centrale se grupează în trei categorii.

Magistrala de adrese transportă adresa locației de memorie sau a portului de intrare-ieșire care urmează să fie accesat. Ea este unidirecțională: adresa pleacă întotdeauna de la unitatea centrală spre memorie, niciodată invers. Lățimea acestei magistrale, adică numărul ei de linii, determină capacitatea maximă de memorie care poate fi adresată, pentru că fiecare combinație de biți desemnează o locație. O magistrală de n linii poate adresa 2^n locații; de exemplu, o magistrală de 32 de biți poate adresa 2^{32} locații, adică 4 GB. Aceasta este o limită fizică absolută: niciun artificiu de programare nu poate face o unitate centrală să acceseze mai multă memorie decât îi permit liniile de adresă, ci doar să comute între blocuri, prin tehnici de paginare.

Magistrala de date transportă informația propriu-zisă între unitatea centrală și memorie sau periferice. Ea este bidirecțională, pentru că datele merg în ambele sensuri: la citire dinspre memorie spre unitatea centrală, la scriere invers. Lățimea ei indică volumul de date transferat simultan; o magistrală de 64 de biți transferă 64 de biți într-un singur ciclu. Această lățime este cea care dă, prin convenție, numele arhitecturii: un microcontroler „pe 8 biți”, cum este ATmega328P, are magistrala de date internă de opt biți, deci lucrează natural cu octeți. O adunare pe 16 biți nu este imposibilă, dar se face în două operații, cu propagarea transportului, deci durează aproximativ de două ori mai mult.

Magistrala de control transportă semnalele care coordonează operațiile: comanda de citire, comanda de scriere, semnalul de ceas, semnalele de validare, cererile de întrerupere. Ea este în general bidirecțională, pentru că nu numai unitatea centrală trimite comenzi, ci și perifericele îi semnalează evenimente. Fără magistrala de control, celelalte două ar fi mute: adresa și data ar fi prezente pe fire, dar nimeni nu ar ști în ce moment sunt valabile și în ce sens circulă.

2.7 Arhitectura generală a unui microcontroler

Modul în care aceste magistrale sunt așezate față de memorie definește arhitectura sistemului. Există două soluții clasice, iar diferența dintre ele are consecințe directe asupra vitezei.

2.7.1 Arhitectura von Neumann

În *arhitectura von Neumann* există o singură memorie, comună pentru date și pentru instrucțiuni, și o singură magistrală care leagă memoria de unitatea centrală. Sistemul conține trei unități de bază: unitatea centrală, care preia instrucțiunile din memorie, le decodifică și le execută; memoria, care stochează atât instrucțiunile cât și datele folosite în timpul execuției; și unitățile de intrare-ieșire, care permit interacțiunea cu dispozitivele externe.

Execuția unei instrucțiuni cere în această arhitectură cel puțin două etape distincte. Mai întâi se extrage instrucțiunea din memorie, la adresa dată de numărătorul de program, și se decodifică. Apoi, dacă instrucțiunea are nevoie de un operand aflat în memorie, acesta se aduce printr-un al doilea acces, se execută operația și se salvează rezultatul în regiștri sau în memorie.

Problema este că cele două accese folosesc aceeași magistrală și nu pot avea loc simultan. Instrucțiunea și data se îmbulzesc pe același drum, iar unitatea centrală așteaptă. Fenomenul se numește *gâtuirea von Neumann* și este limita fundamentală a acestei arhitecturi. Avantajul ei, în schimb, este flexibilitatea: fiindcă programul și datele stau în același spațiu, un program poate să își modifice propriul cod sau să încarce în memorie un program nou, ceea ce face arhitectura potrivită pentru calculatoare de uz general.

2.7.2 Arhitectura Harvard

În *arhitectura Harvard* spațiile de memorie pentru instrucțiuni și pentru date sunt separate fizic, iar fiecare are magistralele lui proprii, folosite exclusiv pentru instrucțiuni sau exclusiv pentru date. Consecința imediată este că aducerea unei instrucțiuni și accesul la o dată pot avea loc în același timp, pe drumuri diferite.

Avantajele sunt trei și decurg unele din altele. Performanța crește, pentru că magistralele nu se mai concurează. Transferul de date se îmbunătățește, pentru că lățimea fiecărei magistrale poate fi aleasă independent: pe AVR, magistrala de program are 16 biți, cât o instrucțiune, în timp ce magistrala de date are 8 biți, cât un octet. Și, cel mai important, devine posibilă execuția cvasiparalelă: în timp ce o instrucțiune se execută, următoarea este deja adusă din memoria de program. Din acest motiv, pe un AVR, majoritatea instrucțiunilor se execută într-un singur ciclu de ceas.

Figura 2.2 arată această organizare în forma ei minimală: nucleul cu unitatea aritmetică și fișierul de regiștri, legat prin magistrale distincte de memoria de program și de memoria de date, iar prin aceasta din urmă și de blocul perifericelor.

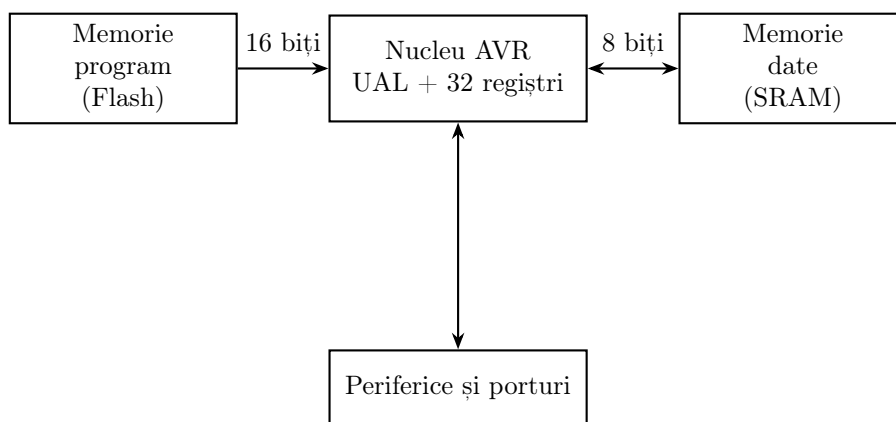


Figura 2.2 Schema-bloc a nucleului AVR cu magistrale separate pentru program și pentru date, specifică arhitecturii Harvard.

2.7.3 Execuția cvasiparalelă și conceptul de pipeline

Ideea de a suprapune etapele de execuție se numește *pipeline*, prin analogie cu o bandă de asamblare. Mai multe instrucțiuni se află simultan în lucru, dar fiecare într-o altă etapă. În timp ce prima instrucțiune se execută, a doua se decodifică și a treia se extrage din memorie. Rezultatul este că, deși fiecare instrucțiune traversează în continuare toate etapele, o instrucțiune se *termină* la fiecare ciclu, iar viteza aparentă crește proporțional cu numărul de etaje.

Tabela 2.2 Suprapunerea etapelor de execuție într-un pipeline cu trei etaje.

	Ciclul 1	Ciclul 2	Ciclul 3	Ciclul 4
Instrucțiunea 1	Extrage	Decodează	Execută	
Instrucțiunea 2		Extrage	Decodează	Execută
Instrucțiunea 3			Extrage	Decodează

Implementarea unui pipeline este însă complexă și cere o sincronizare precisă între etaje, pentru că apar situații în care banda de asamblare trebuie oprită. Aceste situații se numesc *hazarduri* și sunt de trei feluri:

1. *Hazard de date*: o instrucțiune are nevoie de rezultatul unei instrucțiuni anterioare care nu s-a terminat încă. Pipeline-ul trebuie să aștepte sau să trimită rezultatul înainte, prin căi speciale.
2. *Hazard de control*: apare la instrucțiunile de ramificare, unde rezultatul condiției nu este cunoscut în momentul în care instrucțiunea următoare a fost deja extrasă. Dacă saltul se face, instrucțiunile aduse degeaba trebuie abandonate.
3. *Hazard structural*: două instrucțiuni au nevoie de aceeași resursă hardware în același moment.

Pe AVR, pipeline-ul are doar două etaje, aducere și execuție, iar consecința hazardului de control se vede direct în tabelele de sincronizare din foaia de catalog: o instrucțiune de salt durează doi cicluri dacă saltul se face și un singur ciclu dacă nu se face, pentru că în primul caz instrucțiunea deja adusă trebuie aruncată și înlocuită. Este un detaliu mărunț, dar el explică de ce durata unei bucle scrise cu grijă nu este perfect constantă.

2.8 Setul de instrucțiuni

Setul de instrucțiuni reprezintă mulțimea comenzilor de bază pe care un microcontroler le înțelege. El este definit în siliciu, prin structura decodicatorului, și constituie granița dintre hardware și software: orice program, indiferent de limbajul în care a fost scris, ajunge în cele din urmă să fie o succesiune de instrucțiuni din acest set.

Indiferent de familie, orice microcontroller are patru clase de instrucțiuni. Instrucțiunile *aritmetice și logice* efectuează calculele: adunare, scădere, comparație, operații logice, deplasări. Instrucțiunile *de decizie* implementează o bifurcație în graful programului, adică saltul condiționat sau necondiționat. Instrucțiunile *de transfer al datelor* mută valori între regiștri, între regiștri și memorie, între regiștri și porturi. Instrucțiunile *de test și la nivel de bit* verifică sau modifică biți individuali, fără să atingă restul octetului; ele sunt semnătura microcontrolerelor și explică de ce comanda unui pin se poate face într-o singură instrucțiune.

Tabela 2.3 Exemple de instrucțiuni AVR, grupate pe clase.

Clasa	Exemple	Efect
Aritmetice, logice	ADD, SUB, AND, EOR, LSL	operează asupra a doi regiștri și actualizează indicatorii din SREG
Transfer de date	MOV, LDI, LD, ST	mută un octet între regiștri sau între registru și memorie
Decizie	RJMP, BRNE, RCALL, RET	modifică numărătorul de program, condiționat sau nu
La nivel de bit	SBI, CBI, SBIC	pun pe 1, pun pe 0 sau testează un singur bit dintr-un registru de intrare-ieșire

2.8.1 RISC față de CISC

După filosofia de proiectare a setului, microcontrolerele se împart în două familii.

Un procesor *RISC* (*reduced instruction set computer*) recunoaște și execută doar operații de bază, puține și uniforme. Operațiile complicate se obțin prin combinarea celor elementare, deci un program RISC este mai lung în număr de instrucțiuni. În schimb, fiindcă instrucțiunile sunt simple și au toate aceeași dimensiune, ele pot fi decodificate rapid și executate, în majoritate, într-un singur ciclu de ceas. Aceasta face execuția previzibilă, ceea ce contează enorm în sistemele în timp real, unde trebuie să știm exact cât durează o secvență.

Un procesor *CISC* (*complex instruction set computer*) dispune de un set bogat, cu peste optzeci de instrucțiuni, atât de bază cât și complexe. Multe dintre ele sunt specializate pentru funcții anume, optimizate pentru anumite tipuri de sarcini, și diferă mult între ele ca format și ca durată. Un program CISC este mai scurt în număr de instrucțiuni, ceea ce conta mult când memoria era scumpă, dar durata fiecărei instrucțiuni variază și decodificarea este complicată.

Familia AVR, din care face parte ATmega328P, este de tip RISC. Setul ei cuprinde 131 de instrucțiuni, aproape toate codificate pe 16 biți și aproape toate executate într-un singur ciclu de ceas. Aceasta este alegerea care explică, mai bine decât orice altceva, raportul favorabil între viteză și consum al acestor circuite.

2.9 Arhitectura ATmega328P

Punând laolaltă noțiunile de până acum, putem descrie concret circuitul cu care vom lucra în tot restul cărții. ATmega328P folosește o *arhitectură Harvard modificată*. Cuvântul „modificată” este important: spațiile de program și de date sunt separate, ca în Harvard, dar există o cale prin care unitatea centrală poate citi constante din memoria de program, prin instrucțiunea LPM și registrul de adresă Z. Fără această portiță, un tabel de constante ar trebui copiat în SRAM la pornire, ceea ce ar consuma inutil o resursă rară.

Circuitul are trei memorii, cu roluri complet distincte, prezentate în Figura 2.3.

Memoria de program este o memorie Flash de 32 KB, folosită pentru stocarea codului. În timpul execuției, instrucțiunile sunt citite de aici și executate de nucleul AVR. Fiindcă o instrucțiune AVR ocupă un cuvânt de 16 biți, cei 32 KB înseamnă 16384 de cuvinte, adică 2^{14} locații; de aceea numărătorul de program are exact 14 biți. Memoria Flash este împărțită în două zone: secțiunea de aplicație, unde stă programul nostru, și secțiunea de încărcare (*bootloader*), o zonă mică de la adresele superioare care conține un program capabil să își rescrie singur restul memoriei. Pe plăcile Arduino, bootloader-ul este cel care primește programul nou prin legătura serială și îl scrie în secțiunea de aplicație; el ocupă câțiva sute de octeți, care nu mai sunt disponibili pentru programul utilizatorului.

Memoria de date propriu-zisă este o memorie SRAM de 2 KB, folosită pentru stocarea datelor temporare, a variabilelor și a celorlalte informații necesare execuției programului. Ea suportă cinci moduri diferite de adresare, iar accesul intern la ea se face în doi cicluri de ceas ai unității centrale. Spațiul de adrese al datelor nu conține însă numai SRAM: la adresele cele mai mici se află, în ordine, cei 32 de regiștri generali, apoi 64 de regiștri de intrare-ieșire, apoi 160 de regiștri de intrare-ieșire estinși, și abia după ei începe memoria SRAM propriu-zisă. Această așezare unitară este comodă, pentru că permite accesarea regiștrilor perifericelor cu aceleași instrucțiuni ca o variabilă obișnuită, ceea ce în C se traduce prin faptul că `PORTB` arată ca o variabilă și se folosește ca o variabilă.

Memoria EEPROM de 1 KB este destinată datelor care trebuie să rămână salvate chiar și după oprirea alimentării microcontrolerului: parametri de calibrare, setări alese de utilizator, contoare de ore de funcționare, numere de serie. Ea are propriul spațiu de adrese, separat, și nu se accesează direct, ci prin trei regiștri dedicați, printr-o procedură de câțiva pași.

2.9.1 De ce SRAM-ul mic este constrângerea reală

Un student care vede cifrele 32 KB, 2 KB și 1 KB este tentat să creadă că memoria de program este cea care limitează aplicația, pentru că acolo scriem cod. În practică lucrurile stau invers. Codul compilat pentru un AVR este dens, iar o aplicație de complexitate rezonabilă, cu citiri de senzori, comunicație serială și un afișaj, încapă confortabil în 32 KB. Cei 2 KB de SRAM sunt însă epuizați cu o ușurință surprinzătoare.

Motivul este că în SRAM stau simultan mai multe lucruri: toate variabilele globale, toate variabilele locale ale funcțiilor active, toate tampoanele de comunicație și, peste toate, stiva, care crește dinspre celălalt capăt. Un singur tablou de 500 de numere întregi pe 16 biți

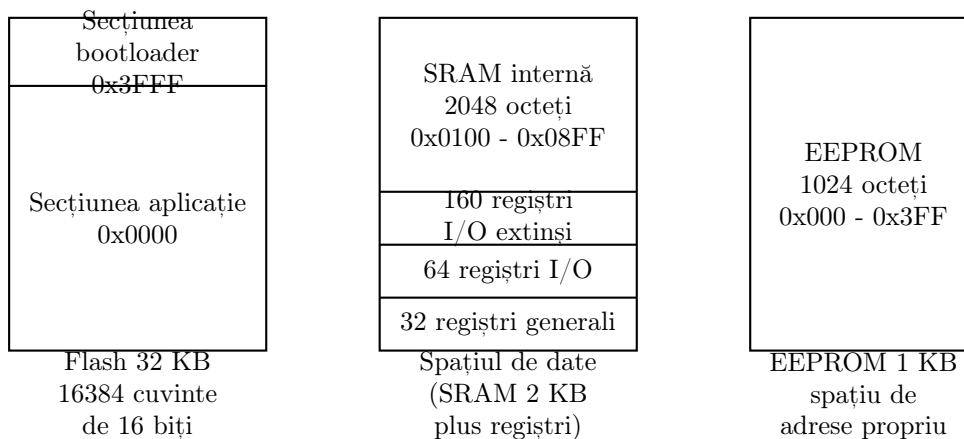


Figura 2.3 Harta celor trei memorii ale unui ATmega328P: Flash pentru program, spațiul de date cu regiștri și SRAM, respectiv EEPROM cu spațiu de adrese propriu.

ocupă 1000 de octeți, adică jumătate din memorie. Un șir de caractere scris direct în cod, de tipul `Serial.print("Temperatura masurata este: ")`, este copiat implicit în SRAM la pornire, împreună cu toate celelalte șiruri constante din program. Într-un proiect cu multe mesaje de diagnostic, aceste șiruri pot ocupa singure câteva sute de octeți.

Simptomul unei memorii SRAM epuizate nu este un mesaj de eroare, pentru că nu există nimeni care să îl dea: compilatorul nu poate ști cât de adânc se va apela stiva în timpul execuției. Simptomul este comportamentul haotic descris la stivă. De aceea, obișnuința sănătoasă este să privim mereu, după compilare, raportul de utilizare a memoriei și să tratăm o ocupare a SRAM-ului de peste aproximativ 75 la sută ca pe un semnal de alarmă.

Exemplu

Un program de achiziție păstrează ultimele 300 de măsurători ale unui senzor, fiecare măsurătoare fiind un număr întreg pe 16 biți, și mai folosește un tampon de recepție serială de 128 de octeți. Să se estimeze memoria SRAM rămasă liberă pe un ATmega328P.

Rezolvare. Tabloul de măsurători ocupă $300 \times 2 = 600$ octeți. Tamponul de recepție ocupă 128 de octeți. Împreună, cele două structuri consumă 728 de octeți. Din cei 2048 de octeți disponibili rămân $2048 - 728 = 1320$ de octeți.

Din acești 1320 de octeți trebuie însă scăzute și celelalte consumuri: variabilele globale ale bibliotecilor folosite, care pe o placă Arduino cu comunicație serială depășesc frecvent 200 de octeți, șirurile de caractere constante copiate în SRAM și, mai ales, spațiul pe care stiva îl va cere în cel mai defavorabil lanț de apeluri, inclusiv la intrarea în rutinele de tratare a întreruperilor. O rezervă de câteva sute de octeți pentru stivă este minimul rezonabil.

Concluzia este că marja reală nu este de 1320 de octeți, ci de câteva sute. Dacă cerința ar crește la 800 de măsurători, tabloul singur ar ocupa 1600 de octeți și proiectul nu ar mai încăpea; soluția ar fi fie reducerea rezoluției la 8 biți acolo unde precizia permite, fie păstrarea unei medii mobile în locul întregului istoric, fie folosirea unei memorii externe.

2.10 Circuitul oscilator

Toate operațiile descrise până acum se petrec în pași discreți, iar pașii sunt dați de un semnal periodic numit *ceas* (*clock*). Circuitul oscilator furnizează acest semnal, necesar pentru sincronizarea proceselor interne. Fără el, microcontrolerul nu este defect, ci pur și simplu îngheață: numărătorul de program nu mai avansează, instrucțiunile nu mai sunt aduse, nimic nu se mai întâmplă. Un microcontroler căruiua i s-a oprit ceasul consumă foarte puțin, ceea ce este de altfel principiul modurilor de economie de energie.

2.10.1 Oscilatorul cu cristal de cuarț

Un *oscilator cu cuarț* generează un semnal electric cu o frecvență foarte stabilă. Elementul activ este o lamelă de cuarț tăiată după o orientare cristalografică precisă. Cuarțul este piezoelectric: aplicarea unei tensiuni îl deformează mecanic, iar deformarea mecanică generează o tensiune. Lamela are, ca orice corp elastic, o frecvență proprie de rezonanță mecanică, determinată de dimensiunile ei. Închizând bucla între deformare și tensiune printr-un amplificator, sistemul oscilează exact la această frecvență proprie.

Avantajul decisiv este că frecvența proprie a unei lamele de cuarț depinde de dimensiuni, nu de tensiune sau de temperatură, cel puțin nu în măsură apreciabilă. Stabilitatea tipică a unui cristal obișnuit este de ordinul a câteva zeci de părți per milion, adică o eroare relativă de ordinul 10^{-5} . În montaj, cristalul se leagă între pinii OSC1 și OSC2 ai microcontrolerului, împreună cu două condensatoare de valoare mică, de ordinul zecilor de picofarazi, legate la masă. Aceste condensatoare de sarcină nu sunt decorative: ele închid bucla de reacție și aduc cristalul exact la frecvența nominală, care este specificată de producător pentru o anumită capacitate de sarcină.

O variantă practică este *rezonatorul ceramic* încapsulat, care conține în același corp elementul rezonant și cele două condensatoare, deci se leagă cu trei terminale și ocupă mai puțin loc. Precizia lui este mai slabă decât a unui cristal de cuarț, de ordinul a câteva părți la mie, dar suficientă pentru multe aplicații.

2.10.2 Oscilatorul RC

Un *oscilator RC* folosește încărcarea și descărcarea unui condensator printr-o rezistență, cu un comparator care basculează la atingerea unor praguri. Perioada rezultată este proporțională cu produsul RC . Soluția este ieftină și se integrează complet în siliciu, fără componente externe, dar nu are o precizie foarte bună: frecvența generată variază cu tensiunea de alimentare, cu temperatura și cu dispersia de fabricație a rezistențelor și capacităților integrate.

ATmega328P are un asemenea oscilator RC intern, cu frecvența nominală de 8 MHz, calibrat din fabrică. Circuitul este livrat cu o configurație implicită în care acest oscilator intern este sursa de ceas, divizată suplimentar cu opt, deci microcontrolerul pornește la 1 MHz dacă nu i se spune altceva. Sursa de ceas se alege prin biții de configurație permanentă numiți *siguranțe* (*fuses*), care se scriu la programare și nu pot fi modificați din programul obișnuit.

Tabela 2.4 Comparație între sursele de ceas uzuale pentru un microcontroler.

Sursă	Precizie tipică	Observații
Oscilator RC intern	câteva procente	fără componente externe, variază cu tensiunea și temperatura, eliberează doi pini
Rezonator ceramic	câteva părți la mie	trei terminale, condensatoare integrate, cost mic
Cristal de cuarț	zeci de părți per milion	necesită două condensatoare de sarcină, soluția standard când timpul contează

2.10.3 Când contează precizia

Diferența dintre câteva procente și câteva zeci de părți per milion pare academică până în momentul în care aplicația măsoară sau comunică.

Dacă microcontrolerul doar aprinde un LED cu o întârziere de o secundă, o eroare de frecvență de trei la sută înseamnă că secunda durează în realitate 0,97 sau 1,03 secunde, ceea ce nimeni nu observă. Dacă însă microcontrolerul ține evidența orei, aceeași eroare înseamnă o abatere de aproape o oră pe zi, ceea ce face ceasul inutilizabil.

Cazul cel mai sever este comunicația serială asincronă. Într-un cadru UART, receptorul nu primește niciun semnal de ceas; el se sincronizează pe frontul de start și apoi eșantionează biții la intervale calculate cu propriul lui ceas. Dacă cele două ceasuri diferă prea mult, eroarea se acumulează pe lungimea cadrului și, la ultimii biți, receptorul eșantionează în locul greșit. Marja tolerabilă pentru un cadru standard de zece biți este de ordinul a două procente pentru abaterea totală dintre emițător și receptor. Un oscilator RC intern necalibrat, care poate devia cu mai multe procente pe domeniul de temperatură, se află exact la limita acestei marje, iar rezultatul este binecunoscut: comunicație care merge pe masa de laborator și cade în cameră rece. De aceea plăcile Arduino Uno folosesc un rezonator sau un cristal extern de 16 MHz, chiar dacă asta ocupă doi pini.

2.11 Ciclul de instrucțiune

Semnalul furnizat de oscilator nu comandă direct execuția, ci este folosit pentru a construi *ciclul de instrucțiune*, adică succesiunea completă de operații necesare pentru a duce la capăt o instrucțiune. În arhitecturile clasice, ciclul de instrucțiune se obține din n perioade de ceas egale, notate de obicei Q1, Q2, Q3 și Q4 pentru $n = 4$, fiecare fază rezervată unei etape.

Într-o astfel de organizare, faza Q1 este dedicată aducerii instrucțiunii de la adresa indicată de numărătorul de program și incrementării acestuia. Fazele Q2 și Q3 sunt dedicate decodificării instrucțiunii și identificării operanzilor, ceea ce poate include aducerea unei date din memorie. Faza Q4 este dedicată execuției propriu-zise și salvării rezultatului. Un microcontroler organizat astfel are nevoie de patru perioade de ceas pentru fiecare instrucțiune, deci un ceas de 4 MHz produce un milion de instrucțiuni pe secundă.

Aici se vede câștigul concret al arhitecturii Harvard. Fiindcă memoria de program are magistrala ei proprie, nucleul AVR aduce instrucțiunea următoare în timp ce o execută pe cea curentă. Cele două etape nu se așteaptă una pe alta, ci se suprapun, iar rezultatul este că majoritatea instrucțiunilor AVR se termină într-un singur ciclu de ceas. Raportul între frecvența ceasului și numărul de instrucțiuni executate pe secundă devine astfel aproximativ unu.

Exemplu

Un ATmega328P funcționează cu un cristal extern de 16 MHz. Să se estimeze numărul de instrucțiuni executate pe secundă și durata unei instrucțiuni simple.

Rezolvare. La 16 MHz, perioada ceasului este

$$T = \frac{1}{16 \cdot 10^6 \text{ Hz}} = 62,5 \text{ ns.}$$

Deoarece majoritatea instrucțiunilor AVR se execută într-un singur ciclu de ceas, o instrucțiune simplă, cum ar fi o adunare între doi regiștri, durează 62,5 ns, iar microcontrolerul execută aproximativ 16 milioane de instrucțiuni pe secundă, adică 16 MIPS.

Estimarea este o limită superioară. Instrucțiunile care accesează memoria SRAM cer doi cicluri, deci 125 ns; instrucțiunile de salt condiționat cer unul sau doi cicluri, după cum saltul se face sau nu; înmulțirea hardware cere doi cicluri. Un program real, cu apeluri de funcții și accese la memorie, atinge în practică o valoare mai mică, dar ordinul de mărime rămâne acesta.

Un calcul util în practică: o buclă care comută un pin folosind instrucțiuni la nivel de bit are nevoie de câteva cicluri pe iterație, deci frecvența maximă a semnalului generat prin program pe un pin este de ordinul a câteva sute de kilohertzi. Pentru semnale mai rapide sau mai precise nu se folosește programul, ci modulele hardware de numărare, care lucrează independent de nucleu.

Această previzibilitate este una dintre calitățile cele mai prețioase ale microcontrolerelor pe opt biți. Pe un procesor modern de calculator, cu mai multe nivele de memorie cache, execuție speculativă și un sistem de operare care comută între procese, durata unei secvențe de cod nu poate fi prezisă exact. Pe un AVR ea poate fi calculată prin numărarea ciclurilor din foaia de catalog, iar rezultatul este valabil la fiecare execuție. Acesta este motivul pentru care asemenea circuite se folosesc încă în aplicații cu cerințe stricte de timp real, chiar dacă puterea lor de calcul este modestă.

2.12 Alimentarea microcontrolerului

Un microcontroler are nevoie de o tensiune de alimentare continuă, stabilă și curată. Familia din care face parte ATmega328P acceptă un domeniu larg, aproximativ de la 1,8 V la 5,5 V, ceea ce permite alimentarea directă din baterii. Domeniul de tensiune și frecvența maximă de lucru sunt însă legate între ele: tranzistoarele comută mai lent la tensiune mică, deci la 16 MHz circuitul are nevoie de cel puțin 4,5 V, în timp ce la 1,8 V frecvența maximă

garantată scade la câțiva megahertzi. A alimenta un ATmega328P la 3,3 V și a-l ceasa la 16 MHz este o abatere de la specificație care poate funcționa pe masa de laborator și poate ceda la temperaturi extreme.

Cea mai simplă schemă de alimentare de la rețea folosește un transformator coborâtor, o punte redresoare și un stabilizator liniar integrat de tipul LM7805. Transformatorul coboară tensiunea la aproximativ 9 V alternativi, puntea o redresează, un condensator electrolitic de câteva zeci de microfarazi netezește pulsațiile, iar stabilizatorul furnizează la ieșire 5 V continui. Tensiunea aplicată la intrarea stabilizatorului trebuie să fie mai mare decât cea de ieșire cu o marjă de câțiva volți, uzual între 7 V și 24 V continui.

Alegerea capsulei se face după curentul cerut. Pentru curenți până la aproximativ 1 A se folosește capsula TO-220, prevăzută cu o aripioară metalică, obligatoriu cu radiator suplimentar. Pentru curenți mici, până la aproximativ 100 mA, este suficientă capsula mică TO-92. Motivul acestei diferențe este că un stabilizator liniar nu transformă tensiunea, ci o disipă: puterea pierdută este produsul dintre căderea de tensiune pe el și curentul care îl străbate. Cu 12 V la intrare, 5 V la ieșire și 500 mA sarcină, disipația este de $(12-5) \cdot 0,5 = 3,5$ W, adică mai mult decât puterea utilă livrată consumatorului. Pentru randamente mai bune se folosesc stabilizatoare cu comutație, care însă injectează zgomot în alimentare și cer filtrare mai atentă.

2.12.1 De ce condensatoarele de decuplare nu sunt opționale

Pe schemele din foile de catalog apare, lângă fiecare pereche de pini de alimentare și masă, câte un condensator ceramic de 100 nF. Aceste *condensatoare de decuplare* sunt sistematic omise de începători, pentru că montajul pare să meargă și fără ele. Este o falsă economie, iar mecanismul merită înțeles.

Un microcontroller nu consumă curent uniform. La fiecare front de ceas, zeci de mii de tranzistoare comută simultan și, pentru câteva nanosecunde, circuitul cere un vârf de curent mult mai mare decât media. Sursa de alimentare nu poate răspunde la o asemenea cerere, pentru că între ea și circuit se află fire și trasee. Orice conductor are o inductanță parazită, iar tensiunea pe o inductanță este proporțională cu viteza de variație a curentului, $u = L \cdot di/dt$. Un vârf de curent foarte scurt înseamnă un di/dt foarte mare, deci o cădere de tensiune apreciabilă chiar pe firele de alimentare. Rezultatul este că tensiunea la pinul circuitului scade brusc la fiecare comutare.

Condensatorul de decuplare rezolvă problema mutând sursa de energie mai aproape. Fiind legat direct între pinul de alimentare și pinul de masă, la câțiva milimetri de capsulă, el constituie un mic rezervor de sarcină care furnizează instantaneu vârful de curent, fără să mai treacă prin inductanța traseelor. Sursa reîncarcă apoi condensatorul, lent, între două vârfuri.

Consecințele lipsei lui sunt greu de diagnosticat, tocmai pentru că sunt intermitente: resetări aparent aleatorii, citiri analogice zgomotoase, comunicație serială cu erori rare, comportament care se schimbă când atingi placa cu mâna. De aceea regula practică este fără excepție: câte un condensator ceramic de 100 nF la fiecare pereche VCC-GND, cât mai

aproape posibil de pinii circuitului, plus un condensator electrolitic de ordinul a 10 microfarazi la intrarea alimentării pe placă, pentru variațiile mai lente. Pe ATmega328P această regulă se aplică atât perechii VCC-GND, cât și perechii AVCC-GND care alimentează partea analogică; pinul AREF cere și el propriul condensator, pentru că zgomotul de pe tensiunea de referință se regăsește direct în rezultatul conversiei analog-numerice.

Observație

Un condensator de decuplare așezat la doi centimetri de circuit, pe partea cealaltă a plăcii, își pierde aproape complet eficacitatea: inductanța traseului până la el anulează avantajul. La montajul cu componente prin gaură, condensatorul se lipește direct între pinii de alimentare și de masă ai soclului. Poziționarea contează la fel de mult ca valoarea.

2.13 Un exemplu de conectare

Am parcurs până acum toate blocurile din interiorul microcontrolerului. Rămâne întrebarea practică: ce trebuie legat la un ATmega328P scos din pungă pentru ca el să funcționeze.

Răspunsul este surprinzător de scurt. Pentru funcționarea unui microcontroler este nevoie, în esență, de două lucruri: o sursă de alimentare stabilizată și un modul de sincronizare, adică o sursă de ceas. Restul pinilor deservesc aplicația, nu circuitul.

Concret, montajul minimal cuprinde alimentarea de 5 V aplicată la pinii VCC și AVCC, împreună cu masa la pinii GND corespunzători și cu condensatoarele de decuplare de 100 nF lângă fiecare pereche. Urmează sursa de ceas: fie cristalul de 16 MHz legat între pinii OSC1 și OSC2 cu cele două condensatoare de sarcină de aproximativ 22 pF spre masă, fie nimic, dacă se folosește oscilatorul intern configurat prin siguranțe. Mai este necesar pinul de resetare, notat RESET, care este activ pe nivel coborât și trebuie ținut la tensiunea de alimentare printr-o rezistență de ridicare de ordinul a 10 k Ω ; lăsat în aer, el captează zgomot și provoacă resetări aleatorii. Un buton între acest pin și masă permite resetarea manuală. În sfârșit, dacă se folosește convertorul analog-numeric, pinul AREF primește propriul condensator de decuplare.

Exemplu 2.2 Programul minimal care verifica un montaj nou

```

1 // Verificare rapida a unui montaj cu ATmega328P.
2 // Daca LED-ul de pe PB5 clipeste regulat , alimentarea si ceasul
3 // functioneaza , iar programul a fost incarcat corect .
4
5 int main(void)
6 {
7     DDRB |= (1 << PB5);           // PB5 configurat ca iesire
8
9     while (1) {
10         PORTB ^= (1 << PB5);      // inverseaza starea pinului
11         for (volatile long i = 0; i < 200000L; i++) {
```

```

12         ; // intarziere aproximativa , prin bucla goala
13     }
14 }
15     return 0;
16 }

```

Programul de mai sus este cel mai util instrument de depanare pentru un montaj nou. Dacă LED-ul clipește, atunci alimentarea este bună, ceasul oscilează, resetarea nu se activează singură și programul a ajuns în memorie. Dacă LED-ul stă aprins fix, cel mai probabil microcontrollerul este blocat în resetare. Dacă nu se întâmplă nimic, problema este la alimentare sau la ceas. Dacă LED-ul clipește, dar de câteva ori mai rar sau mai des decât se aștepta, atunci ceasul funcționează la altă frecvență decât cea presupusă, ceea ce înseamnă că siguranțele nu sunt configurate pentru sursa de ceas dorită.

Cuvântul `volatile` din declarația contorului merită o notă. Fără el, compilatorul constată că bucla nu produce niciun efect vizibil și o elimină complet la optimizare, iar întârzierea dispăre. Cuvântul `volatile` îi interzice această optimizare, obligându-l să citească și să scrie efectiv variabila la fiecare iterație. Aceeași problemă apare la orice variabilă modificată într-o rutină de tratare a întreruperilor și citită în programul principal, motiv pentru care vom reveni asupra ei.

În fine, o observație de metodă care încheie capitolul. Nicio carte nu poate înlocui foaia de catalog a circuitului folosit. Pentru conectarea fiecărui microcontroller trebuie consultată documentația producătorului, care dă domeniile de tensiune, curenții maximi pe pin, valorile recomandate ale componentelor externe și schema de referință. Ceea ce am descris aici este structura comună, care se regăsește, cu nume și cifre diferite, la orice microcontroller: biți grupați în regiștri, regiștri care comandă periferice, memorii cu roluri distincte, o unitate centrală care execută instrucțiuni în ritmul unui ceas și o alimentare curată care ține totul în viață. Cine a înțeles această structură o va recunoaște imediat în orice foaie de catalog.

Întrebări de verificare

1. Explicați de ce electronica digitală folosește două nivele logice și nu zece, și ce câștigă un microcontroller industrial din această alegere.
2. Valoarea unui registru este `0x6D`. Scrieți-o în binar și în zecimal și precizați care biți sunt activi.
3. De ce scrierea hexazecimală este preferată în foile de catalog față de cea zecimală? Justificați prin relația dintre bazele 2 și 16.
4. Care este diferența funcțională între un registru de control și unul de stare? Dați câte un exemplu de operație greșită care rezultă din confundarea lor.
5. Explicați linie cu linie ce face secvența `DDRB |= (1 << PB5);` urmată de `PORTB &= ~(1 << PB5);` și de ce nu se folosește în loc o atribuire directă.

6. Comparați memoriile SRAM, Flash și EEPROM din punctul de vedere al volatilității, al granularității ștergerii și al numărului de cicluri de scriere. Pentru fiecare, indicați un tip de dată pe care ați plasa-o acolo.
7. Explicați de ce arhitectura Harvard permite execuția majorității instrucțiunilor AVR într-un singur ciclu de ceas, în timp ce o arhitectură von Neumann echivalentă are nevoie de mai mulți.
8. Ce este un hazard de control într-un pipeline și cum se explică prin el faptul că o instrucțiune de salt condiționat durează pe AVR unul sau doi cicluri?
9. Un ATmega328P este ceasat la 8 MHz. Calculați perioada ceasului, durata aproximativă a unei instrucțiuni simple și numărul de instrucțiuni executate pe secundă.
10. Un program folosește două tablouri de câte 400 de octeți și un tampon serial de 64 de octeți. Estimați memoria SRAM rămasă pe un ATmega328P și explicați de ce cifra obținută supraestimează marja reală.
11. Comparați oscilatorul RC intern cu un cristal extern de cuarț și arătați, pe exemplul comunicației seriale asincrone, în ce situație diferența de precizie devine hotărâtoare.
12. Explicați, pornind de la relația $u = L \cdot di/dt$, de ce lipsa condensatoarelor de decuplare produce resetări aparent aleatorii și de ce poziția fizică a acestor condensatoare contează.
13. Enumerați componentele strict necesare pentru ca un ATmega328P să funcționeze și explicați rolul rezistenței de ridicare de pe pinul RESET.

3. Programarea unui microcontroler

- Ce înseamnă a programa un microcontroler
- Etapele programării: de la codul sursă la biții din Flash
- Formatul fișierului Intel HEX
- Structura memoriei Flash: bootloader și zona de aplicație
- Programarea modulelor Arduino
- Programarea prin ISP cu programator dedicat
- Limbaje de programare
- Depanarea unui microcontroler
- Greșeli tipice și cum se recunosc
- Întrebări de verificare

3.1 Ce înseamnă a programa un microcontroler

Studentul care deschide pentru prima dată mediul Arduino, scrie trei linii de cod și apasă butonul de încărcare trăiește un moment de magie: câteva secunde mai târziu, un LED de pe placă începe să clipească. Între apăsarea butonului și clipirea LED-ului se petrec însă o duzină de operații distincte, executate de tot atâtea programe diferite, iar niciuna dintre ele nu este vizibilă pe ecran. Scopul acestui capitol este să deschidă cutia neagră: să arate ce se întâmplă cu textul scris de programator, prin ce forme trece el, ce fișiere se nasc pe disc și cum ajung, în cele din urmă, biții aceia în celulele de memorie Flash ale microcontrolerului.

A *programa* un microcontroler înseamnă, la nivelul cel mai concret, a modifica starea unor celule de memorie nevolatilă aflate în interiorul cipului, astfel încât, la următoarea alimentare, unitatea centrală să găsească acolo o secvență de instrucțiuni pe care le poate executa. Nu există nimic mai mult decât atât. Microcontrolerul nu „înțelege” limbajul C, nu știe ce este o funcție și nu are noțiunea de fișier. El are un registru numărător de program care indică o adresă în memoria Flash, citește de acolo un cuvânt de 16 biți, îl interpretează ca instrucțiune, o execută și trece mai departe. Tot lanțul de unelte descris în acest capitol există pentru un singur motiv: pentru a traduce un text lizibil pentru om într-un șir de cuvinte de 16 biți lizibile pentru unitatea centrală, și pentru a depune acel șir la adresele potrivite.

Aici apare prima diferență majoră față de programarea unui calculator personal. Pe un PC, programul rezultat din compilare rămâne un fișier pe disc; sistemul de operare îl încarcă

în memoria RAM atunci când utilizatorul îl lansează, îi rezolvă legăturile cu bibliotecile partajate, îi dă un spațiu de adrese virtual și îl pornește. Microcontrollerul nu are nici sistem de operare, nici disc, nici încărcător de programe. Programul trebuie să se afle deja în memoria nevolatilă, la adresa exactă la care unitatea centrală îl va căuta, iar adresele acelea sunt fizice, nu virtuale. De aceea procesul se numește *programare* și nu *instalare*: nu copiem un fișier undeva, ci înscrinem permanent o structură de biți într-o memorie.

A doua diferență ține de locul unde se lucrează. Programul este scris, compilat și transformat pe un calculator gazdă, care are cu totul altă arhitectură decât ținta. Compilatorul care rulează pe un procesor x86 și produce cod pentru un AVR pe 8 biți se numește *compilator încrucișat* (engl. *cross-compiler*), iar întreg setul de unelte care îl însoțește formează un *lanț de unelte încrucișat* (engl. *toolchain*). Pentru AVR acest lanț se numește **avr-gcc** și cuprinde, printre altele, preprocesorul, compilatorul propriu-zis, asamblorul, editorul de legături și utilitarele de conversie a formatelor de fișier. La capătul lanțului se află o componentă hardware, *interfața de programare*, singura care atinge fizic microcontrollerul: un programator dedicat sau, în cazul plăcilor Arduino, un simplu cablu USB și un mic program care locuiește deja în cip.

Privind lucrurile de sus, drumul are deci două jumătăți bine separate. Prima este pur software și se petrece integral pe calculatorul gazdă: scrierea codului, compilarea și editarea de legături, până la obținerea unui fișier în formatul pe care îl înțelege programatorul. A doua este o operație de transfer și de scriere fizică într-o memorie, urmată de execuția codului pe microcontroller. Capitolul urmează exact această împărțire.

3.2 Etapele programării: de la codul sursă la biții din Flash

Un programator care lucrează într-un mediu integrat vede o singură comandă, „Compilează” sau „Încarcă”. În spatele ei se execută, în ordine, cinci transformări succesive ale codului, fiecare producând un fișier intermediar cu un rol precis, urmate de operația de transfer în memorie. Faptul că mediile integrate șterg fișierele intermediare la sfârșit nu înseamnă că ele nu au existat; înseamnă doar că studentul nu are ocazia să le vadă. Vom parcurge pe rând fiecare etapă, spunând ce primește la intrare, ce produce la ieșire și, mai important, la ce servește. Lanțul complet este rezumat în Figura 3.1.

3.2.1 Preprocesarea

Prima unealtă care atinge codul sursă nu este compilatorul, ci *preprocesorul*. El nu cunoaște limbajul C; pentru el, fișierul este doar text. Sarcina lui este să execute directivele care încep cu `diez`, adică `#include`, `#define`, `#ifdef` și celelalte, și să livreze compilatorului un text din care acestea au dispărut.

Trei operații sunt esențiale. Prima este *includerea*: fiecare directivă `#include` este înlocuită cu conținutul integral al fișierului indicat, recursiv. Aici se explică de ce un program AVR de cincisprezece linii ajunge, după preprocesare, la zeci de mii de linii: antetul `<avr/io.h>` trage după el fișierul specific dispozitivului, `iom328p.h`, care conține definițiile tuturor registrelor și ale tuturor biților lor. A doua este *substituirea macrouilor*: orice apariție a unui

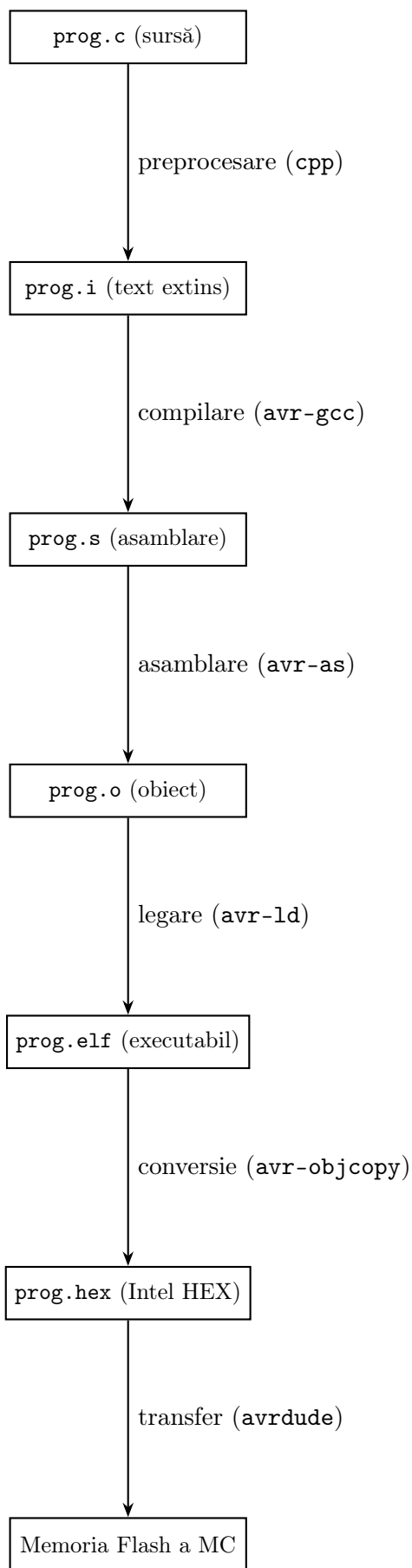


Figura 3.1 Lanțul complet de transformări, de la codul sursă la biții înscriși în memoria Flash. Fiecare săgeată este un program distinct care rulează pe calculatorul gazdă, cu excepția ultimei, care implică și hardware.

nume definit prin `#define` este înlocuită textual cu valoarea lui. Când scriem `PORTB`, preprocesorul înlocuiește numele cu o expresie care desemnează octetul de la adresa fizică a registrului; când scriem `PB5`, îl înlocuiește cu numărul 5. Compilatorul nu vede niciodată numele `PORTB`, ci doar accesul la o adresă. A treia este *compilarea condiționată*: ramurile `#ifdef` neselectate sunt pur și simplu șterse din text. Așa se explică de ce același fișier antet servește pentru zeci de microcontrolere diferite: macroul care indică tipul cipului, transmis de mediul de dezvoltare prin opțiunea `-mmcu`, decide care bloc de definiții rămâne.

Rezultatul este o unitate de traducere, un singur text C autonom, care în lanțul GNU se poate obține explicit cu opțiunea `-E` și are, prin convenție, extensia `.i`. Este util de știut că această etapă există, pentru că multe erori neînțelese de începători se nasc aici: un macro redefinit, un antet inclus de două ori sau un nume de variabilă care coincide cu un macro din biblioteca standard produc mesaje de eroare care par să nu aibă legătură cu linia indicată.

3.2.2 Compilarea propriu-zisă

Compilatorul primește textul preprocesat și produce cod în limbaj de asamblare pentru arhitectura țintă. Este etapa în care se pierde definitiv structura de nivel înalt a programului: bucele, funcțiile, tipurile de date și expresiile aritmetice sunt traduse în instrucțiuni ale unității centrale AVR, iar variabilele sunt repartizate în cele 32 de registre generale sau, dacă nu încap, în memoria SRAM.

Aici lucrează și *optimizatorul*, iar efectul lui asupra codului final este mult mai mare decât își imaginează majoritatea studenților. Compilatorul are dreptul să rearanjeze operațiile, să elimine calcule al căror rezultat nu este folosit, să desfășoare bucle scurte, să înlocuiască o înmulțire cu deplasări și adunări sau să păstreze o variabilă exclusiv într-un registru, fără să o scrie vreodată în memorie. Programele Arduino se compilează implicit cu nivelul de optimizare `-Os`, adică optimizare pentru dimensiune, alegere firească atunci când memoria Flash este resursa critică.

Optimizarea explică și una dintre capcanele clasice ale programării embedded. Dacă o variabilă este modificată de o rutină de tratare a întreruperilor, dar programul principal doar o citește într-o buclă, compilatorul, care analizează numai bucla, constată că nimeni nu modifică variabila și decide să o citească o singură dată, păstrând-o apoi într-un registru. Bucla devine infinită, deși variabila se schimbă în memorie. Remediul este cuvântul cheie *volatile*, care interzice compilatorului această optimizare și îl obligă să recitească variabila din memorie la fiecare utilizare. Nu este un truc, ci o informație pe care programatorul o are și compilatorul nu: că memoria poate fi modificată din afara fluxului de execuție analizat.

La ieșire se obține un fișier text în limbaj de asamblare, cu extensia `.s`. Merită deschis măcar o dată; este cel mai bun mod de a înțelege cât cod mașină generează o linie de C.

3.2.3 Asamblarea

Asamblorul face traducerea unu la unu între mnemonicele limbajului de asamblare și codul mașină binar. Instrucțiunea `sbi 0x05`, 5 devine cuvântul de 16 biți care o codifică; nu există aici nicio decizie de optimizare, doar o corespondență fixă, determinată de manualul

setului de instrucțiuni.

Rezultatul, *fișierul obiect* cu extensia `.o`, nu este însă cod executabil. El conține cod mașină, dar organizat pe secțiuni și, mai ales, incomplet. Secțiunile uzuale sunt `.text` pentru instrucțiuni, `.data` pentru variabilele globale inițializate cu valori nenule, `.bss` pentru cele inițializate implicit cu zero și `.rodata` pentru constante. Incompletitudinea vine din faptul că, la momentul asamblării, nu se știe încă unde vor ajunge lucrurile în memorie: dacă funcția curentă apelează o funcție definită în alt fișier, asamblorul nu cunoaște adresa acesteia și lasă în locul ei un spațiu gol, însoțit de o notă în *tabela de relocări* prin care cere ca cineva să completeze ulterior valoarea. Tot astfel, fișierul obiect poartă o *tabelă de simboluri* cu numele pe care le exportă și cu cele pe care le așteaptă de la alții.

3.2.4 Editarea de legături

Editorul de legături, sau *linker*-ul, este componenta cea mai puțin înțeleasă din lanț și, în același timp, cea care decide harta finală a memoriei. El primește toate fișierele obiect ale proiectului, plus bibliotecile necesare, și are trei sarcini.

Prima este *rezolvarea simbolurilor*: pentru fiecare nume folosit dar nedefinit într-un fișier obiect, caută definiția în celelalte fișiere și în biblioteci. Dacă nu o găsește, produce eroarea binecunoscută „undefined reference”; dacă o găsește de două ori, produce eroarea simetrică, de simbol definit multiplu. A doua este *alocarea adreselor*: linker-ul concatenează secțiunile de același fel din toate obiectele și le așază în spațiul de adrese al dispozitivului, urmând instrucțiunile dintr-un *script de legare* specific familiei. Pentru ATmega328P, secțiunea `.text` începe la adresa 0 a memoriei Flash, iar secțiunile `.data` și `.bss` sunt plasate în SRAM, imediat după zona rezervată registrelor de intrare/ieșire. A treia sarcină este *relocarea*: acum, când adresele sunt cunoscute, toate golurile lăsate de asamblor sunt completate cu valorile corecte.

Linker-ul mai adaugă un lucru pe care programatorul nu l-a scris niciodată: codul de pornire, numit convențional `crt` (engl. *C runtime*). Acesta ocupă primele adrese din Flash, instalează tabela de vectori de întrerupere, inițializează indicatorul de stivă la vârful memoriei SRAM, copiază conținutul secțiunii `.data` din Flash în SRAM, umple cu zero secțiunea `.bss` și abia apoi apelează funcția `main`. Această copiere merită reținută: valorile inițiale ale variabilelor globale trebuie păstrate în memorie nevolatilă, deci ocupă spațiu și în Flash, și în SRAM. Vom regăsi observația când vom discuta despre epuizarea memoriei SRAM.

Rezultatul este un fișier executabil în format ELF, cu extensia `.elf`. El conține codul complet relocat, dar și o cantitate importantă de informație care nu are ce căuta în microcontroler: numele simbolurilor, corespondența dintre adresele mașină și liniile de cod sursă, tipurile variabilelor. Această informație este esențială pentru depanare, motiv pentru care fișierul `.elf` nu trebuie aruncat: fără el, un depanator nu ar putea spune că microcontrolerul se află „în funcția `citesteADC`, linia 12”.

3.2.5 Conversia în format Intel HEX

Un programator de circuite nu are nevoie de simboluri și de informație de depanare; el are nevoie de un răspuns la o singură întrebare: ce octet trebuie scris la ce adresă. De aceea din fișierul ELF se extrag doar secțiunile care ajung efectiv în Flash, adică `.text` și imaginea inițială a lui `.data`, și se transcriu în formatul *Intel HEX*. Operația se face cu utilitarul `avr-objcopy`, iar rezultatul este fișierul `.hex`.

Trebuie subliniat că această etapă nu compilează și nu traduce nimic: codul mașină este deja gata în fișierul ELF. Este o simplă reambalare, dintr-un container complex într-un format textual minimal. Din același ELF se poate extrage separat, cu aceeași unealtă, conținutul destinat memoriei EEPROM, care se salvează într-un fișier cu extensia `.eep` și se încarcă printr-o operație distinctă.

Tabela 3.1 Fișierele produse de lanțul de unelte AVR și rolul fiecăruia.

Extensie	Produs de	Conținut și rol
<code>.c</code> , <code>.cpp</code> , <code>.ino</code>	programator	Codul sursă, text lizibil pentru om
<code>.h</code> <code>.i</code>	programator preprocesor	Declarații și definiții incluse în surse Sursa cu includerile rezolvate și macro- crourele substituite
<code>.s</code>	compiler	Cod în limbaj de asamblare pentru AVR
<code>.o</code>	asamblor	Cod mașină pe secțiuni, cu adrese ne- rezolvate
<code>.a</code> <code>.elf</code>	arhivator linker	Bibliotecă, colecție de fișiere obiect Executabil complet relocat, cu sim- boluri și informație de depanare
<code>.hex</code>	<code>objcopy</code>	Perechi adresă-octeți, în text, pentru programator
<code>.eep</code>	<code>objcopy</code>	Conținutul inițial al memoriei EE- PROM
<code>.map</code>	linker	Raport textual cu adresa fiecărui simbol

3.2.6 Transferul în memorie și verificarea

Ultima etapă este singura care implică hardware. Un program de pe calculatorul gazdă citește fișierul `.hex`, îl împarte în pagini și le trimite, printr-un protocol serial, către o entitate care are capacitatea de a scrie în memoria Flash a microcontrollerului. În lumea AVR acest program se numește `avrdude`, prescurtare de la *AVR Downloader Uploader*, și este folosit atât de mediul Arduino, cât și de mediile profesionale.

Scrierea în Flash nu este o operație banală. Memoria Flash este organizată pe *pagini*, iar unitatea minimă de scriere este pagina, nu octetul. Pentru ATmega328P, o pagină are 64 de cuvinte, adică 128 de octeți, iar cele 32 de kiloocteți de Flash sunt împărțiți în 256

de pagini. Mai mult, tehnologia Flash permite trecerea unui bit din 1 în 0 prin scriere, dar nu și invers; readucerea biților în 1 se face numai prin *ștergere*, tot la nivel de pagină sau pentru întregul cip. De aceea secvența corectă este: ștergere, apoi încărcarea paginii într-un registru tampon, apoi comanda de scriere a paginii. Programatorul se ocupă de toată această coregrafie, dar programatorul uman trebuie să știe că ea există, pentru că îi explică durata operației și faptul că memoria Flash suportă un număr finit de cicluri de ștergere-scriere, de ordinul zecilor de mii, cifră mai mult decât suficientă pentru dezvoltare, dar relevantă dacă cineva ar vrea să folosească Flash-ul ca memorie de date rescrisă frecvent.

După scriere se recomandă *verificarea*: programatorul recitește conținutul memoriei și îl compară cu fișierul sursă. Un rezultat diferit indică fie o alimentare insuficientă, fie o frecvență de programare prea mare, fie o memorie uzată. În fine, o comandă de resetare pune numărătorul de program la zero și execuția începe.

Observație

Etapele de la preprocesare până la generarea fișierului `.hex` nu ating microcontrolerul și nu depind de el; ele pot fi executate fără ca placa să fie conectată. Doar ultima etapă cere hardware. Este o distincție utilă la depanarea erorilor: dacă mesajul apare înainte de bara de progres a încărcării, problema este în cod sau în configurare, nu în cablu, în placă sau în port.

3.3 Formatul fișierului Intel HEX

Fișierul `.hex` este ultimul obiect pe care îl poate inspecta un om înainte ca programul să dispară în cip, și este singurul din lanț care are un format suficient de simplu pentru a fi citit cu ochiul liber. Formatul se numește *Intel HEX* și a fost definit în anii 1970 pentru transferul programelor către memorii nevolatile prin legături seriale lente și nesigure. Toate deciziile de proiectare se explică prin acest context: fișierul este text pur ASCII, ca să treacă prin orice terminal, fiecare linie este autonomă și poartă propria adresă, iar fiecare linie are o sumă de control.

Faptul că fiecare linie își poartă adresa are o consecință practică importantă: fișierul nu trebuie să acopere continuu memoria. El poate conține un bloc la adrese mici, un salt și un alt bloc departe, exact ca în cazul unei plăci Arduino unde aplicația stă la începutul Flash-ului, iar bootloader-ul la sfârșit. Ordinea liniilor nu este, în principiu, obligatoriu crescătoare, deși uneltele uzuale o generează astfel.

3.3.1 Structura unei linii

Fiecare linie, numită formal *înregistrare* (engl. *record*), începe cu caracterul două puncte și continuă cu perechi de cifre hexazecimale, fiecare pereche reprezentând un octet. Structura generală, în notația folosită la curs, este `:BBaaAATTDD...DDCC`, unde câmpurile au următoarea semnificație:

- BB - numărul de octeți de date de pe linia curentă, pe un octet. Uneltele GNU folosesc uzual 16 octeți pe linie, adică 10 în hexazecimal.

- **aaAA** - adresa de la care se scriu octeții, pe doi octeți, cu octetul cel mai semnificativ primul.
- **TT** - tipul înregistrării, care spune cum trebuie interpretat restul liniei.
- **DD...DD** - octeții de date propriu-ziși, adică exact codul mașină generat de compilator; sunt atâția câți anunță câmpul **BB**.
- **CC** - suma de control a liniei, pe un octet.

Câmpul de tip merită o discuție separată, pentru că el transformă un format altfel rigid într-unul flexibil. Tipul 00 desemnează o înregistrare de date obișnuită: „scrie octeții următori începând de la adresa indicată”. Tipul 01 marchează sfârșitul de fișier și apare o singură dată, la final. Tipul 04 este *adresa liniară extinsă* și rezolvă o limitare fundamentală a formatului: câmpul de adresă are doar doi octeți, deci poate exprima cel mult 64 de kiloocteți. O înregistrare de tip 04 nu conține cod, ci cei doi octeți superiori ai adresei, care se combină cu adresele pe 16 biți ale liniilor următoare. Pentru ATmega328P, cu 32 de kiloocteți de Flash, o astfel de înregistrare nu este necesară; ea apare însă obligatoriu în fișierele pentru ATmega2560, care are 256 de kiloocteți.

Tabela 3.2 Tipurile de înregistrare Intel HEX întâlnite în practica AVR.

Tip	Denumire	Rol
00	Date	Octeți de program, de scris la adresa indicată
01	Sfârșit de fișier	Ultima linie; nu conține date
02	Adresă de segment	Moștenire din arhitectura x86, nefolosită la AVR
04	Adresă liniară extinsă	Furnizează octeții superiori de adresă, pentru memorii peste 64 KB
05	Adresă de start liniară	Punctul de intrare; ignorat de programatoarele AVR

3.3.2 Suma de control

Ultimul octet al fiecărei linii protejează linia împotriva erorilor de transmisie. El se calculează simplu: se adună toți octeții liniei, de la numărătorul de octeți până la ultimul octet de date inclusiv, se păstrează numai octetul inferior al sumei și se ia complementul față de doi al acestuia. Altfel spus, se inversează toți biții și se adună unu. Proprietatea utilă a acestei alegeri este că suma tuturor octeților liniei, inclusiv a sumei de control, trebuie să dea zero pe opt biți. Receptorul nu are deci nevoie să refacă niciun calcul complicat: adună tot ce a primit și verifică dacă rezultatul este nul.

Exemplu

Se dă linia :0A000000182800000000000000288E. Să se identifice câmpurile și să se verifice suma de control.

Rezolvare. Despărțim linia pe câmpuri, urmând structura :BBaaAATTDD...DDCC. Numărătorul de octeți este 0A, adică 10 octeți de date. Adresa este 0000, deci datele se scriu de la începutul memoriei. Tipul este 00, adică date de program. Urmează cei 10 octeți de date, 18 28 00 00 00 00 00 00 28, iar ultimul octet, 8E, este suma de control.

Verificăm mai întâi consistența: după cele patru cifre ale antetului de lungime, adresă și tip, rămân $10 \times 2 = 20$ de cifre hexazecimale de date, plus două pentru suma de control. Numărăm și găsim exact atât, deci linia este completă.

Adunăm acum toți octeții, fără suma de control:

$$0A + 00 + 00 + 00 + 18 + 28 + 00 \cdot 7 + 28 = 72_{16} = 114_{10}.$$

Octetul inferior al sumei este deci $72_{16} = 0111\ 0010_2$. Complementul față de doi se obține inversând biții și adunând unu:

$$\overline{0111\ 0010} = 1000\ 1101, \quad 1000\ 1101 + 1 = 1000\ 1110 = 8E_{16}.$$

Rezultatul coincide cu octetul din fișier, deci linia este corectă. Verificarea alternativă: $72 + 8E = 100_{16}$, al cărui octet inferior este zero, exact cum cere proprietatea sumei de control.

Ultima linie a oricărui fișier Intel HEX este :00000001FF. Se citește imediat: zero octeți de date, adresa 0000 fără semnificație, tipul 01 adică sfârșit de fișier, iar suma de control este complementul lui 01, adică FF. Absența acestei linii face ca programatorul să refuze fișierul, considerându-l trunchiat.

3.3.3 Un exemplu de la începutul unui program AVR

Merită privită și prima linie a unui fișier .hex generat pentru ATmega328P, pentru că ea arată ceva despre organizarea internă a programului. O astfel de linie arată tipic așa:

Exemplu 3.1 Prima linie dintr-un fișier hex pentru ATmega328P

```
1 :100000000C9434000C943E000C943E000C943E0082
```

Numărătorul este 10, adică 16 octeți; adresa este 0000; tipul este 00. Datele sunt 0C 94 34 00, apoi 0C 94 3E 00 repetat de trei ori. Octeții se citesc doi câte doi, în ordine inversă, pentru că AVR stochează cuvintele cu octetul inferior primul: perechea 0C 94 este cuvântul 940C, care este codul instrucțiunii de salt absolut jmp, iar perechea următoare dă adresa destinație. Ceea ce vedem la începutul memoriei este deci *tabela de vectori de întrerupere*: o listă de instrucțiuni de salt, câte una pentru fiecare sursă de întrerupere, în ordinea fixată de fabricant. Prima intrare, la adresa 0, este vectorul de resetare și sare la codul de pornire; celelalte, care în acest program nu sunt folosite, sar toate în același loc, la rutina implicită care doar resetează microcontrolerul.

Verificarea sumei de control se face la fel ca mai sus: suma octeților de antet și de date

este 894_{10} , al cărei octet inferior este $7E_{16}$, iar complementul față de doi al lui $7E$ este 82 , exact octetul final al liniei.

3.4 Structura memoriei Flash: bootloader și zona de aplicație

Am spus la începutul capitolului că a programa înseamnă a scrie în Flash. Rămâne o întrebare pe care studentul o pune, pe drept, imediat ce înțelege lanțul: cine scrie? Memoria Flash se află în interiorul cipului, iar terminalele lui nu expun magistrala de memorie. Există două răspunsuri, și ele corespund celor două moduri de programare pe care le vom compara în acest capitol. Ori un dispozitiv extern preia controlul cipului și îl comandă printr-o interfață serială dedicată, ori microcontrollerul se programează singur, executând un program care se află deja în el.

A doua variantă pare paradoxală, dar este perfect posibilă, pentru că familia AVR dispune de instrucțiunea `spm`, prescurtare de la *store program memory*, care scrie o pagină în memoria Flash. Un program care rulează pe microcontroller poate deci modifica memoria de program a aceluiași microcontroller. Programul care face asta se numește *bootloader* sau *încărcător de pornire*.

3.4.1 Cele două secțiuni ale memoriei Flash

Pentru ca mecanismul să fie sigur, memoria Flash a lui ATmega328P este împărțită în două secțiuni, ilustrate în Figura 3.2. *Secțiunea de aplicație* ocupă partea de jos a spațiului de adrese, începe la adresa zero și conține programul scris de utilizator. *Secțiunea de pornire* ocupă partea de sus, se termină la adresa maximă și conține bootloader-ul. Distincția nu este doar de convenție: ea este impusă hardware. Instrucțiunea `spm` se execută cu efect numai atunci când numărătorul de program se află în secțiunea de pornire. Cu alte cuvinte, un program de aplicație obișnuit nu poate rescrie Flash-ul nici din greșeală, nici intenționat; numai codul aflat în zona de bootloader are acest privilegiu.

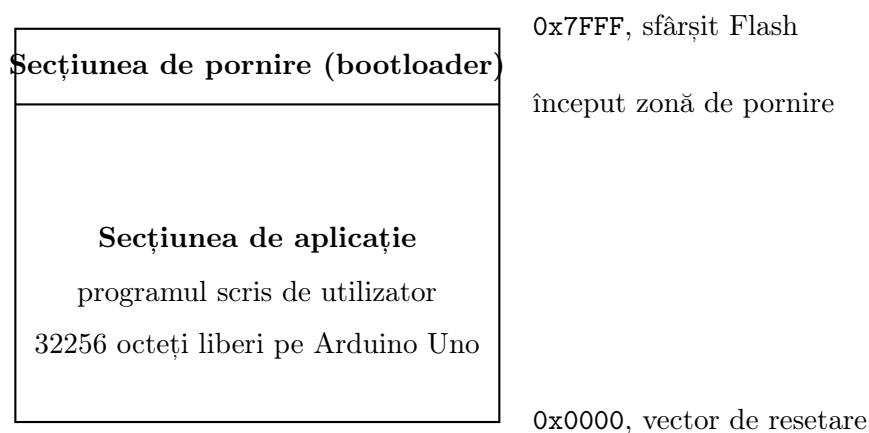


Figura 3.2 Împărțirea celor 32 de kiloocteți de memorie Flash ai lui ATmega328P în secțiune de aplicație și secțiune de pornire. Proporțiile nu sunt la scară: pe o placă Arduino Uno bootloader-ul ocupă o cincizecime din memorie.

Dimensiunea secțiunii de pornire nu este fixă, ci se alege dintre patru valori posibile, prin doi biți de configurare permanentă numiți *siguranțe* (engl. *fuses*), notați `BOOTSZ1` și `BOOTSZ0`. Valorile disponibile sunt 256, 512, 1024 sau 2048 de cuvinte, adică 512 octeți, 1, 2 sau 4 kiloocteți. O a treia siguranță, `BOOTRST`, decide de unde începe execuția după resetare: dacă nu este activată, numărătorul de program pornește de la adresa zero, deci direct din aplicație; dacă este activată, pornește de la prima adresă a secțiunii de pornire, deci din bootloader. Aceste siguranțe nu se pot modifica prin bootloader, ci numai printr-un programator extern, ceea ce este firesc: altfel un program greșit ar putea dezactiva chiar mecanismul care permite încărcarea următorului program.

3.4.2 Ce face un bootloader

Comportamentul unui bootloader, oricare ar fi el, urmează același tipar. La alimentare sau la resetare, controlul ajunge în secțiunea de pornire. Bootloader-ul inițializează interfața de comunicație pe care o folosește și așteaptă o scurtă perioadă, de ordinul sutelor de milisecunde, un mesaj de sincronizare de la calculatorul gazdă. Dacă mesajul sosește, bootloader-ul intră în dialog: primește comenzi, recepționează pagini de cod, le scrie în secțiunea de aplicație cu instrucțiunea `spm` și, la sfârșit, execută un salt la adresa zero, dând controlul programului proaspăt încărcat. Dacă mesajul nu sosește în intervalul de așteptare, bootloader-ul renunță și sare direct la aplicație, adică la ultimul program încărcat. Aceasta este explicația secunde de întârziere pe care o observă oricine a alimentat o placă Arduino: nu este o lentoare a plăcii, este fereastra în care bootloader-ul ascultă.

Interfața de comunicație folosită depinde de implementare. Pe plăcile Arduino este interfața serială asincronă UART, dar există bootloadere care folosesc I2C, SPI, USB nativ, CAN sau chiar legături radio, în cazul actualizărilor prin aer. Principiul rămâne neschimbat; se schimbă doar driverul de comunicație.

Avantajul acestei soluții este considerabil. Ea elimină necesitatea unui programator dedicat, care costă bani și presupune acces fizic la conectorul de programare. Ea permite actualizarea programului unui echipament deja montat și sigilat, printr-un port pe care acesta îl are oricum. Și, în cazul plăcilor de dezvoltare, ea permite ca legătura serială folosită pentru încărcarea programului să fie aceeași cu cea folosită pentru mesajele de depanare, ceea ce simplifică enorm munca începătorului.

3.4.3 Prețul plătit și ce se pierde dacă bootloader-ul dispare

Nimic nu este gratuit. Bootloader-ul consumă spațiu în Flash, iar spațiul acela nu mai este disponibil pentru aplicație. Bootloader-ul Optiboot, folosit pe Arduino Uno, ocupă 512 octeți, motiv pentru care mediul Arduino raportează 32256 de octeți disponibili, în loc de 32768. Bootloaderele mai vechi, folosite pe plăcile Duemilanove și pe unele clone, ocupau 2 kiloocteți, adică o șaisprezecime din memorie. Al doilea cost este timpul de pornire: fereastra de așteptare întârzie lansarea aplicației după fiecare resetare, ceea ce într-un produs industrial poate fi inacceptabil. Al treilea cost este mai subtil: aplicația nu poate folosi vectorii de întrerupere în intervalul în care rulează bootloader-ul, iar unele bootloadere lasă

periferice inițializate într-o stare pe care aplicația trebuie să o cunoască.

Ce se întâmplă dacă bootloader-ul este șters, de exemplu prin programarea cipului cu un programator extern, care execută o ștergere completă înainte de scriere? Microcontrollerul continuă să funcționeze perfect, executând aplicația nou scrisă. Ceea ce se pierde este posibilitatea de a-l mai programa prin cablul serial. De acum înainte, orice modificare va cere din nou un programator extern, până când bootloader-ul este rescris. Aceasta este funcția comenzii „Burn Bootloader” din mediul Arduino: ea nu face altceva decât să programeze, printr-un programator ISP, imaginea bootloader-ului în secțiunea de pornire și să seteze siguranțele `BOOTRST` și `BOOTSZ` astfel încât cipul să pornească de acolo.

Observație

Un microcontroller ATmega328P cumpărat de la distribuitor nu are bootloader și nu poate fi programat serial. Pentru a-l folosi pe o placă de tip Arduino trebuie mai întâi programat o dată, prin ISP, cu bootloader-ul și cu siguranțele potrivite. Din același motiv, un cip înlocuit după o defecțiune nu va răspunde la încărcarea prin USB decât după această operație inițială.

3.5 Programarea modulelor Arduino

Pe o placă Arduino Uno, cele descrise până acum se combină într-un lanț pe care merită să îl urmărim pas cu pas, pentru că fiecare verigă explică o observație practică familiară oricui a lucrat cu placa.

Placa are, pe lângă microcontrollerul principal ATmega328P, un al doilea circuit care se ocupă de conversia dintre USB și interfața serială. Calculatorul gazdă vede acest al doilea circuit ca pe un port serial virtual; liniile lui de transmisie și recepție sunt legate la terminalele `RXD` și `TXD` ale microcontrollerului principal. Nimic din acest montaj nu programează nimic; el doar transportă octeți.

3.5.1 Resetarea automată

Prima problemă practică este că bootloader-ul ascultă doar imediat după resetare. Pe plăcile foarte vechi utilizatorul trebuia să apese butonul de resetare exact în momentul în care mediul de dezvoltare începea încărcarea, ceea ce era o sursă constantă de frustrare. Soluția actuală este elegantă: o linie de control a portului serial virtual, numită `DTR`, este legată la terminalul de resetare al microcontrollerului printr-un condensator de valoare mică. Când programul de pe calculator deschide portul serial, el comută linia `DTR`, iar condensatorul transmite doar frontul, generând un impuls scurt care resetează microcontrollerul. Placa se resetează singură exact în clipa potrivită, iar bootloader-ul pornește și ascultă.

Acest detaliu explică un comportament care nedumerește mulți studenți: dacă deschid monitorul serial al mediului de dezvoltare, placa se resetează și programul o ia de la capăt. Nu este o eroare, este efectul aceleiași linii `DTR`.

3.5.2 Dialogul dintre avrdude și bootloader

Pe calculatorul gazdă rulează **avrdude**, care primește ca parametri tipul cipului, tipul programatorului, portul serial, viteza de comunicație și fișierul de încărcat. O comandă tipică, pe care mediul Arduino o construiește automat și o afișează dacă se activează ieșirea detaliată, arată astfel:

Exemplu 3.2 Comanda avrdude generată de mediul Arduino pentru o placă Uno

```
1 avrdude -C avrdude.conf -p atmega328p -c arduino \  
2 -P /dev/ttyUSB0 -b 115200 -D \  
3 -U flash:w:prog.hex:i
```

Parametrul **-p** indică microcontrolerul țintă, **-c** indică protocolul de programare, **-P** portul, **-b** viteza în biți pe secundă, iar **-U** descrie operația: memoria **flash**, acțiunea **w** adică scriere, fișierul și formatul **i** adică Intel HEX. Opțiunea **-D** dezactivează ștergerea completă a cipului, tocmai pentru a nu distruge bootloader-ul aflat în secțiunea de pornire.

Protocolul folosit în dialog se numește *STK500*, după numele plăcii de evaluare Atmel pentru care a fost definit inițial. El este un protocol simplu de comenzi și răspunsuri: gazda cere sincronizarea, întreabă semnătura cipului, trimite adresa paginii, trimite conținutul paginii, cere scrierea, apoi trece la pagina următoare. La final poate cere recitirea, pentru verificare. Bootloader-ul Optiboot implementează un subset al acestui protocol, suficient pentru scrierea în Flash; el nu poate, de exemplu, să scrie siguranțele sau să citească memoria EEPROM, tocmai pentru a rămâne în cei 512 octeți alocați.

Viteza de comunicație trebuie să corespundă exact între cele două capete, pentru că este fixată în codul bootloader-ului. Optiboot folosește 115200 de biți pe secundă; bootloaderele mai vechi foloseau 57600. Aceasta este cauza cea mai frecventă a erorii **not in sync**, care apare atunci când mediul de dezvoltare are selectat un alt tip de placă decât cel conectat efectiv.

3.5.3 O placă Arduino în rol de programator

O placă Arduino poate juca și rolul programatorului extern pentru un al doilea microcontroler. Se încarcă în ea o schiță specială, numită convențional **ArduinoISP**, care implementează pe partea de gazdă protocolul STK500, iar pe partea de ieșire generează semnalele interfeței seriale sincrone SPI necesare programării. Se leagă apoi cele patru semnale, plus alimentarea, la cipul țintă. Această configurație este exact ceea ce folosește comanda de scriere a bootloader-ului atunci când utilizatorul nu are un programator comercial. Este și modul obișnuit de a repara o placă al cărei bootloader a fost șters din greșeală.

3.6 Programarea prin ISP cu programator dedicat

Programarea *în sistem* (engl. *In-System Programming*, ISP), numită uneori și ICSP, *In-Circuit Serial Programming*, este modalitatea de bază, cea care nu depinde de niciun cod aflat deja în microcontroler. Ea folosește interfața serială sincronă SPI a cipului, dar

într-un regim special, activat prin menținerea terminalului de resetare în stare activă. Cât timp microcontrollerul este ținut în resetare, unitatea centrală nu execută nimic, iar logica de programare preia controlul: ea recepționează pe terminalul MOSI instrucțiuni de programare de câte patru octeți și răspunde pe MISO, sincronizată de ceasul aplicat pe SCK. Legăturile sunt ilustrate în Figura 3.3.

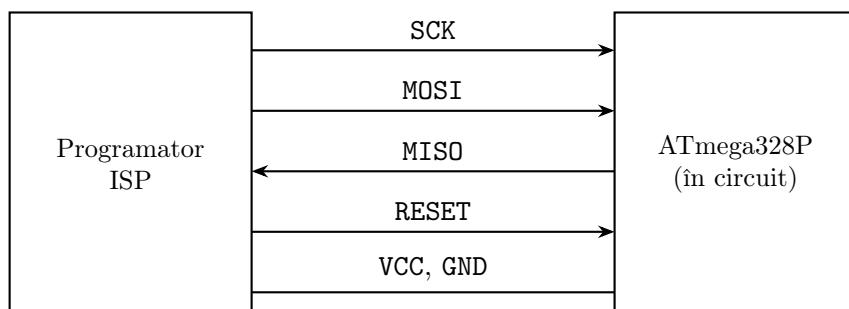


Figura 3.3 Cele șase semnale ale conectorului ISP. Programatorul ține microcontrollerul în resetare și îi comandă direct logica de programare a memoriei.

Prima comandă trimisă este întotdeauna cea de activare a programării; dacă microcontrollerul răspunde cu octetul de confirmare așteptat, dialogul poate continua. Se citește apoi *semnătura* cipului, trei octeți care identifică fabricantul și modelul, iar programatorul verifică dacă tipul selectat de utilizator corespunde cu ce este efectiv conectat. Urmează ștergerea, încărcarea paginilor și verificarea.

O constrângere de care începătorii se lovesc este cea de frecvență. Ceasul SCK aplicat de programator nu poate depăși o pătrime din frecvența de ceas a microcontrollerului, pentru că logica de programare este sincronizată intern cu ceasul sistemului. Un cip nou, care funcționează pe oscilatorul intern divizat, are un ceas de 1 MHz, deci acceptă cel mult 250 kHz pe SCK. Un programator configurat pentru o frecvență mai mare va raporta pur și simplu că nu găsește niciun dispozitiv, deși legăturile sunt corecte. Majoritatea programatoarelor au un comutator sau o opțiune software pentru „ceas lent”, care rezolvă situația.

3.6.1 Când este necesar fiecare mod de programare

Cele două moduri nu se exclud, ci se completează. Programarea prin bootloader este comodă, rapidă și nu cere hardware suplimentar, dar are limite structurale: consumă Flash, întârzie pornirea, nu poate modifica siguranțele și nu poate reface bootloader-ul însuși. Programarea prin ISP este universală, ajunge la orice zonă a memoriei și la orice bit de configurare, dar cere un dispozitiv separat și acces la conectorul de programare. Tabelul 3.3 rezumă comparația.

Pe scurt: în timpul dezvoltării obișnuite se folosește bootloader-ul, pentru că este suficient și rapid. Programatorul ISP devine indispensabil în trei situații: la prima punere în funcțiune a unui cip nou, atunci când trebuie modificată configurația de ceas sau de protecție, și atunci când aplicația are nevoie de fiecare octet de Flash și de o pornire instantanee, caz în care bootloader-ul se elimină deliberat.

Tabela 3.3 Programarea prin bootloader față de programarea prin ISP.

Criteriu	Bootloader (serial)	ISP (SPI)
Hardware necesar	doar cablul USB al plăcii	programator dedicat sau altă placă
Cod prezent în cip	bootloader deja scris	niciunul; funcționează pe cip nou
Spațiu Flash consumat	512 octeți sau mai mult	niciunul
Acces la siguranțe	nu	da
Acces la biții de blocare	nu	da
Poate scrie bootloader-ul	nu	da
Întârziere la pornire	da, fereastra de ascultare	nu
Utilizare tipică	dezvoltare curentă, actualizări în teren	prima programare, reparare, producție

3.7 Limbaje de programare

Un microcontroler poate fi programat, în principiu, în orice limbaj pentru care există un compilator care generează cod pentru arhitectura respectivă. În practică, alegerea este mult mai restrânsă decât pe un calculator, iar motivul este resursa: cu 32 de kiloocteți de Flash și 2 kiloocteți de SRAM, nu există loc pentru o mașină virtuală, pentru un colector de gunoaie sau pentru o bibliotecă standard bogată. Limbajele care domină lumea microcontrolerelor mici sunt, de aceea, cele care se traduc direct și previzibil în cod mașină.

3.7.1 Limbajul de asamblare

Limbajul de asamblare este notația simbolică a setului de instrucțiuni al procesorului. Fiecare linie corespunde exact unei instrucțiuni mașină, iar programatorul controlează personal fiecare registru și fiecare ciclu de ceas.

Avantajele sunt reale și nu trebuie minimalizate. Codul rezultat este cel mai compact posibil și cel mai rapid posibil, pentru că nu conține nimic în plus față de ce a scris autorul. Consumul de resurse este perfect cunoscut: se poate spune, numărând, exact câte cicluri durează o rutină și exact câți octeți ocupă, ceea ce este esențial în bucle de control cu termen strict sau în rutine de întrerupere care trebuie să se încheie într-un timp garantat. Pentru programe mici, de câteva zeci de linii, implementarea este chiar comodă.

Dezavantajele apar odată cu dimensiunea. Un program de câteva mii de linii în asamblare devine foarte greu de citit, de modificat și de verificat, pentru că intenția programatorului nu mai este vizibilă în text; ea trebuie reconstruită din succesiunea operațiilor elementare. Apoi, codul nu este portabil: el depinde nu doar de familia de microcontrolere, ci uneori chiar de modelul exact, pentru că numărul de registre, setul de instrucțiuni și adresele perifericelor

diferă. Astăzi asamblarea se folosește punctual, în rutine scurte și critice, nu pentru aplicații întregi.

3.7.2 Limbajul C

Limbajul C ocupă poziția dominantă în programarea microcontrolerelor și o ocupă pentru motive de fond, nu din obișnuință. C a fost proiectat ca un limbaj apropiat de mașină: tipurile lui corespund unităților naturale de date ale procesorului, operatorii pe biți permit manipularea directă a registrelor, iar pointerii permit accesul la adrese fizice. Un compilator C bun generează, pentru cod scris cu grijă, un rezultat comparabil cu asamblarea scrisă de om, dar în câteva minute în loc de câteva zile.

La aceasta se adaugă portabilitatea. Același algoritm scris în C se poate recompila pentru altă familie de microcontrolere schimbând doar porțiunile care ating direct registrele, adesea izolate într-un strat subțire. Există compilatoare pentru practic orice arhitectură, există o bibliotecă standard, iar pentru domeniile critice există seturi de reguli, cum sunt regulile MISRA, și unelte de analiză statică ce verifică respectarea lor. Prețul este o curbă de învățare abruptă la început, mai ales în privința pointerilor și a gestionării manuale a memoriei, plus lipsa oricărei plase de siguranță: C permite fără să protesteze scrierea în afara unui tablou sau folosirea unui pointer neinițializat, iar rezultatul, pe un microcontroler, este o defecțiune greu de urmărit.

C++ adaugă peste C organizarea în clase, capsularea și reutilizarea codului. Bibliotecile Arduino sunt scrise în C++ și beneficiază vizibil de asta: un obiect `Serial` sau un obiect `Servo` grupează starea și operațiile într-o entitate coerentă. Costul este că unele mecanisme, cum sunt alocarea dinamică, tratarea excepțiilor sau identificarea tipurilor la execuție, consumă memorie în cantități pe care un cip cu 2 kiloocteți de SRAM nu le poate susține; de aceea, în practica embedded, se folosește un subset disciplinat al limbajului, din care aceste mecanisme sunt excluse. Limbajele interpretate sau cu execuție gestionată, precum Python în variantele lui reduse, sau limbaje mai noi precum Rust, sunt folosite pe microcontrolere cu mult mai multe resurse decât ATmega328P.

3.7.3 Ce ascund bibliotecile Arduino

Mediul Arduino a devenit popular pentru că ascunde registrele în spatele unor funcții cu nume citibile. Este o alegere pedagogică bună la început și o alegere costisitoare mai târziu, iar studentul trebuie să înțeleagă exact ce se schimbă.

Să comparăm cele două moduri de a comuta un pin de ieșire. Varianta cu funcții Arduino este imediată de citit.

Exemplu 3.3 Comutarea unui pin cu funcțiile bibliotecii Arduino

```
1 void setup() {  
2     pinMode(13, OUTPUT);  
3 }  
4
```

```

5 void loop() {
6   digitalWrite(13, HIGH);
7   digitalWrite(13, LOW);
8 }

```

Varianța cu acces direct la registre este mai puțin evidentă, dar spune exact ce se întâmplă în cip.

Exemplu 3.4 Aceeași operație, scrisă direct în registrele portului B

```

1 #include <avr/io.h>
2
3 int main(void) {
4   DDRB |= (1 << PB5);           /* PB5 devine iesire */
5   while (1) {
6     PORTB |= (1 << PB5);        /* nivel ridicat */
7     PORTB &= ~(1 << PB5);      /* nivel coborat */
8   }
9 }

```

Diferența de comportament este nulă: pe placa Uno, pinul 13 al conectorului este chiar terminalul PB5. Diferența de cost este însă mare. Instrucțiunea generată pentru `PORTB |= (1 << PB5)` este `sbi`, care setează un bit într-un registru de intrare/ieșire și se execută în două cicluri de ceas. Apelul `digitalWrite(13, HIGH)` face, în schimb, o serie de operații suplimentare: traduce numărul de pin al conectorului în portul și bitul corespunzător, citind două tabele aflate în memoria Flash, verifică dacă pinul respectiv este atașat unui canal de modulație în durată și, dacă da, îl detașează, apoi dezactivează întreruperile, modifică registrul și le reactivează. Rezultatul este o funcție de ordinul zecilor de cicluri de ceas, în locul celor două ale instrucțiunii `sbi`.

Exemplu

Un ATmega328P funcționează la 16 MHz. Să se estimeze frecvența maximă a semnalului dreptunghiular obținut prin comutarea unui pin în buclă, în cele două variante, presupunând că un apel `digitalWrite` consumă aproximativ 50 de cicluri de ceas, iar instrucțiunea `sbi` sau `cbi` consumă 2 cicluri.

Rezolvare. Perioada unui ciclu de ceas este

$$T_{clk} = \frac{1}{16 \cdot 10^6 \text{ Hz}} = 62,5 \text{ ns}.$$

O perioadă completă a semnalului cere o comutare în sus și una în jos. În varianta cu bibliotecă, aceasta înseamnă $2 \times 50 = 100$ de cicluri, la care se adaugă ciclurile buclei; ignorându-le, obținem

$$T_{Arduino} \approx 100 \times 62,5 \text{ ns} = 6,25 \text{ } \mu\text{s}, \quad f_{Arduino} \approx 160 \text{ kHz}.$$

În varianta cu acces direct, o perioadă cere $2 \times 2 = 4$ cicluri, deci

$$T_{direct} \approx 4 \times 62,5 \text{ ns} = 250 \text{ ns}, \quad f_{direct} \approx 4 \text{ MHz}.$$

Raportul este de aproximativ 25, adică exact raportul numărului de cicluri. Concluzia practică: pentru aprinderea unui LED sau citirea unui buton, diferența este complet irelevantă, pentru că omul nu percepe microsecunde. Pentru generarea unui semnal, pentru comunicația implementată software sau pentru o rutină de întrerupere care trebuie să se încheie repede, diferența decide dacă aplicația funcționează sau nu.

Concluzia nu este că bibliotecile Arduino sunt de evitat, ci că ele reprezintă un compromis explicit: siguranță și comoditate în schimbul vitezei și al memoriei. Un program bine scris folosește funcțiile de bibliotecă în părțile necritice, unde lizibilitatea contează, și coboară la registre acolo unde contează fiecare ciclu. A înțelege ce face funcția pe care o apelezi este, în embedded, o condiție a controlului asupra propriului program.

3.8 Depanarea unui microcontroler

Depanarea unui program care rulează pe un calculator este o operație confortabilă: programul rulează sub un sistem de operare care îl poate opri oricând, un depanator îi poate citi memoria, iar dacă ceva merge prost, procesul se încheie și restul mașinii continuă. Nimic din toate acestea nu este adevărat pe un microcontroler, iar dificultatea depanării în embedded provine din câteva cauze structurale.

Prima este lipsa de vizibilitate. Microcontrolerul nu are ecran, nu are consolă și nu are sistem de fișiere în care să scrie un jurnal. Singura fereastră spre interiorul lui este cea pe care programatorul o construiește deliberat. A doua este natura sistemului: el interacționează cu lumea fizică în timp real. Un motor care se rotește, un condensator care se încarcă sau un senzor care eșantionează nu se opresc pentru că am pus un punct de întrerupere; oprirea procesorului poate schimba complet comportamentul sistemului sau poate chiar produce pagube. A treia este că multe defecte nu sunt în cod, ci la granița dintre cod și hardware: un fir prost lipit, o alimentare care cade sub tensiunea minimă în momentul pornirii unui motor, o rezistență de ridicare lipsă. Un depanator software nu are nicio șansă să le vadă. A patra este că resursele necesare depanării, memoria de program și pinii, sunt exact resursele care lipsesc.

3.8.1 Depanarea la nivel hardware: debugWIRE, JTAG, SWD

Producătorii au integrat în cipuri module dedicate de depanare, care permit unui dispozitiv extern să controleze execuția fără a folosi resursele programului.

Pentru microcontrolerle AVR mici, inclusiv ATmega328P, mecanismul se numește *debugWIRE* și are o proprietate remarcabilă: folosește un singur fir, chiar terminalul de resetare, pe care îl transformă într-o legătură bidirecțională de comandă. Prin el, dispozitivul extern poate opri procesorul, citi și scrie registrele și memoria, executa instrucțiuni pas cu pas și

fixa puncte de întrerupere. Se activează printr-o siguranță numită *DWEN*, iar aici apare capcana: odată activată, terminalul de resetare nu mai funcționează ca resetare obișnuită, deci programarea prin ISP, care depinde de el, devine imposibilă. Ieșirea din această situație se face printr-o comandă specială a depanatorului, care dezactivează modul; dacă se pierde legătura înainte de asta, cipul trebuie recuperat printr-o programare paralelă la înaltă tensiune, operație care necesită echipament specializat. Mai trebuie știut că plăcile Arduino Uno nu expun comod acest mod, pentru că terminalul de resetare este legat la circuitul de resetare automată.

JTAG este un standard mult mai vechi și mai general, definit inițial pentru testarea plăcilor cu circuite integrate. El folosește patru sau cinci terminale și oferă, pe lângă depanare, posibilitatea de a testa conexiunile de pe placă și de a programa memoria. Este disponibil pe microcontrolerele AVR mari, precum ATmega2560, și pe majoritatea arhitecturilor ARM. *SWD*, *Serial Wire Debug*, este varianta redusă a lui *JTAG*, specifică microcontrolerelor ARM, care obține aceleași funcții cu doar două terminale, economie importantă pe cipuri mici.

Funcțiile oferite de un astfel de modul sunt cele așteptate de la orice depanator: puncte de întrerupere la o anumită linie de cod, execuție pas cu pas, inspecția și modificarea variabilelor și a registrelor, vizualizarea memoriei, inclusiv a stivei, și urmărirea lanțului de apeluri de funcții. Toate acestea funcționează pentru că dispozitivul de depanare citește fișierul `.elf` și știe astfel ce adresă corespunde cărei linii și ce variabilă stă la ce adresă. Numărul punctelor de întrerupere hardware este însă limitat, de regulă la câteva, iar oprirea procesorului rămâne intruzivă.

3.8.2 De ce mesajele pe serială rămân unealta principală

În ciuda existenței acestor mecanisme, cea mai folosită metodă de depanare în lumea microcontrolerelor mici rămâne tipărirea de mesaje pe interfața serială. Explicația nu este comoditatea, ci un cumul de motive solide.

Nu cere hardware suplimentar: pe o placă Arduino, legătura serială există deja, este aceeași cu cea folosită pentru încărcarea programului. Nu oprește execuția, deci nu perturbă interacțiunea cu lumea fizică; sistemul continuă să funcționeze în timp ce raportează. Permite observarea evoluției în timp, ceea ce un punct de întrerupere nu poate face: o valoare care derivează lent sau o întrerupere care apare o dată la o mie de cicluri se văd într-un jurnal, dar nu într-o oprire. Și, nu în ultimul rând, mesajele pot fi lăsate în cod, activate printr-un macro de compilare condiționată, și reactivate mai târziu, când produsul deja funcționează în teren.

Metoda are și limite pe care studentul trebuie să le cunoască. Tipărirea consumă timp: la 9600 de biți pe secundă, transmiterea unui mesaj de treizeci de caractere durează peste treizeci de milisecunde, o eternitate pentru o rutină de întrerupere. Consumă și memorie SRAM, cum vom vedea imediat. Și, cel mai perfid, poate schimba comportamentul programului: un defect cauzat de o sincronizare strânsă poate dispărea atunci când adăugăm tipăriri și poate reapărea când le scoatem. Când suspiciunea cade pe timp, o alternativă mai

bună este comutarea unui pin liber și observarea lui pe osciloscop sau pe analizorul logic: costul este de câteva cicluri, iar rezoluția temporală este excelentă.

Observație

Înainte de a bănuși codul, trebuie verificată alimentarea și trebuie verificat că microcontrollerul chiar rulează. O buclă care aprinde intermitent un LED, plasată la începutul programului, răspunde în trei secunde la întrebarea dacă cipul pornește, dacă ceasul este cel presupus și dacă programul a ajuns efectiv în Flash. Este cel mai ieftin instrument de diagnostic existent.

3.9 Greșeli tipice și cum se recunosc

Există câteva erori care apar cu regularitate la primele proiecte și care au în comun faptul că mesajul de eroare, atunci când există, nu indică deloc cauza reală. Merită tratate explicit, pentru că recunoașterea lor scutește ore de căutare.

3.9.1 Memoria Flash plină

Prima este cea mai simplă, pentru că mediul de dezvoltare o semnalează clar: programul depășește spațiul disponibil. Cauza obișnuită nu este dimensiunea codului scris de student, ci bibliotecile incluse. O bibliotecă grafică pentru un afișor, o bibliotecă de comunicație în rețea sau folosirea funcțiilor de formatare în virgulă mobilă pot adăuga fiecare mii de octeți. Trebuie amintit și că din cei 32768 de octeți, bootloader-ul consumă deja 512.

Soluțiile, în ordinea efortului, sunt: eliminarea bibliotecilor neutilizate, renunțarea la aritmetica în virgulă mobilă acolo unde numerele întregi sunt suficiente, mutarea tabelelor constante în Flash cu atributul `PROGMEM` și, în ultimă instanță, eliminarea bootloader-ului și programarea prin ISP.

3.9.2 Memoria SRAM epuizată

A doua eroare este mult mai insidioasă, pentru că nu produce niciun mesaj. Programul se compilează, se încarcă, pornește și funcționează un timp, după care începe să dea rezultate absurde, să se reseteze singur sau să se blocheze. Cauza este epuizarea celor 2048 de octeți de SRAM.

Mecanismul merită înțeles în detaliu. Memoria SRAM găzduiește trei lucruri: variabilele globale și statice, alocate la compilare în secțiunile `.data` și `.bss`; zona de alocare dinamică, dacă se folosește; și stiva, care crește de la adresa cea mai mare spre adrese mai mici și găzduiește variabilele locale, argumentele funcțiilor și adresele de revenire. Compilatorul cunoaște și raportează dimensiunea primei categorii, dar nu are cum să știe cât de adânc va crește stiva la execuție. Când stiva coboară peste zona variabilelor globale, ea începe să le suprascră, și invers. Nu există nicio unitate de protecție a memoriei care să oprească acest lucru; programul continuă să ruleze cu date corupte.

Cel mai frecvent consumator neașteptat de SRAM sunt șirurile de caractere constante. Într-un program C compilat pentru AVR, un literal de tipul `"Temperatura masurata: "`

este, implicit, copiat din Flash în SRAM de codul de pornire, pentru că arhitectura AVR are spații de adrese separate pentru program și pentru date, iar pointerii obișnuiți indică în SRAM. Douăzeci de astfel de mesaje de câte treizeci de caractere consumă 600 de octeți, adică aproape o treime din memoria disponibilă, fără ca programatorul să fi declarat vreo variabilă. Remediu este macroul `F()` din biblioteca Arduino, respectiv atributul `PROGMEM` și funcțiile din familia `_P` în C simplu; ele păstrează șirul în Flash și îl citesc caracter cu caracter, la nevoie, cu instrucțiunea `lpm`.

Exemplu 3.5 Sir de caractere in SRAM (sus) si acelasi sir pastrat in Flash (jos)

```

1  /* varianta care consuma SRAM */
2  Serial.println("Temperatura masurata: ");
3
4  /* varianta care pastreaza sirul in Flash */
5  Serial.println(F("Temperatura masurata: "));

```

Semnul de alarmă practic este raportul afișat de mediul de dezvoltare după compilare: dacă variabilele globale ocupă peste aproximativ 75% din SRAM, spațiul rămas pentru stivă devine riscant, iar comportamentul aleatoriu este de așteptat.

3.9.3 Ceasul configurat altfel decât presupune codul

A treia eroare tipică se manifestă prin întârzieri greșite și prin comunicație serială ilizibilă. Cauza este o nepotrivire între frecvența reală de ceas a microcontrolerului și valoarea pe care o presupune codul compilat.

Codul nu măsoară timpul; el numără cicluri. Funcția `_delay_ms` din biblioteca `avr-libc` calculează numărul de iterații ale unei bucle de așteptare pornind de la macroul `F_CPU`, definit la compilare. Tot din `F_CPU` se calculează și valoarea înscrisă în registrul divizorului de viteză al interfeței seriale. Dacă `F_CPU` spune 16 MHz, iar cipul funcționează în realitate pe oscilatorul intern de 8 MHz cu divizarea prin 8 activă prin siguranța `CKDIV8`, adică la 1 MHz, atunci toate întârzierile vor fi de șaisprezece ori mai lungi, iar viteza reală a interfeței seriale va fi de șaisprezece ori mai mică decât cea așteptată. Simptomul este un LED care clipește foarte lent și un monitor serial care afișează caractere fără sens.

Situația apare tipic la un cip nou, programat prin ISP, la care nimeni nu a setat siguranțele pentru oscilatorul extern, sau atunci când în mediul de dezvoltare este selectată o placă cu alt cristal decât cel montat. Verificarea este simplă: se programează o buclă care aprinde LED-ul o secundă și îl stinge o secundă și se cronometrează. Dacă perioada observată este de opt sau șaisprezece ori mai mare decât cea așteptată, cauza este ceasul, nu programul.

Exemplu

Un program compilat cu `F_CPU` egal cu 16000000 conține apelul `_delay_ms(500)`. Microcontrolerul funcționează însă pe oscilatorul intern la 1 MHz. Cât durează în realitate întârzierea?

Rezolvare. Compilatorul a calculat numărul de cicluri necesare presupunând frecvența

declarată:

$$N = 0,5 \text{ s} \times 16 \cdot 10^6 \text{ cicluri/s} = 8 \cdot 10^6 \text{ cicluri.}$$

Acest număr este fix, înscris în cod. La execuție, ciclurile se consumă însă la 1 MHz, deci durata reală este

$$t = \frac{8 \cdot 10^6}{1 \cdot 10^6} = 8 \text{ s,}$$

adică de șaisprezece ori mai mult decât cei 0,5 secunde intenționați. Raportul este exact raportul frecvențelor, ceea ce oferă și metoda de diagnostic: se măsoară o întârziere cunoscută și se calculează frecvența reală ca produs dintre frecvența declarată și raportul dintre durata așteptată și cea măsurată.

3.9.4 Alte confuzii frecvente

Merită enumerate pe scurt și celelalte capcane. Selectarea unui alt tip de placă în mediul de dezvoltare duce la o viteză de comunicație greșită cu bootloader-ul și la eroarea de sincronizare descrisă mai sus. Portul serial ocupat de monitorul serial sau de altă aplicație face ca încărcarea să eșueze fără explicație clară. Un pin folosit simultan de aplicație și de interfața de programare, cum sunt terminalele PB3, PB4 și PB5 în cazul ISP, poate împiedica programarea dacă circuitul extern legat la ei încarcă prea mult liniile. În fine, omiterea cuvântului cheie `volatile` la variabilele partajate cu rutinele de întrerupere produce un program care funcționează la nivel de optimizare zero și se blochează la nivelul folosit implicit, ceea ce sugerează greșit o eroare de compilator.

Numitorul comun al tuturor acestor situații este că simptomul se află departe de cauză. Metoda de lucru care ajută cel mai mult este reducerea: se pornește de la cel mai mic program care funcționează sigur, de exemplu clipirea unui LED, și se adaugă câte un element o dată, verificând după fiecare pas. Este mai lent decât căutarea directă a erorii, dar este singura metodă care converge întotdeauna.

Întrebări de verificare

1. Descrieți, în ordine, transformările prin care trece un fișier sursă în C până la înscrierea codului în memoria Flash, precizând pentru fiecare etapă unealta care o execută și fișierul rezultat.
2. Explicați de ce fișierul `.elf` nu poate fi transmis direct programatorului și ce informație se pierde la conversia lui în format Intel HEX.
3. Care sunt cele trei sarcini ale editorului de legături? Explicați ce înseamnă relocarea și de ce ea nu poate fi făcută de asamblor.
4. Calculați suma de control pentru o înregistrare de date de patru octeți, cu adresa 0100 și octeții de date 01 02 03 04, scrieți linia completă și explicați de ce suma tuturor octeților liniei trebuie să dea zero pe opt biți.

5. Explicați de ce un program obișnuit de aplicație nu poate scrie în memoria Flash a propriului microcontroler și ce condiție hardware trebuie îndeplinită pentru ca instrucțiunea `spm` să aibă efect.
6. Comparați programarea prin bootloader cu programarea prin ISP din punctul de vedere al hardware-ului necesar, al spațiului consumat și al accesului la biții de configurare. Ce se pierde dacă bootloader-ul este șters și în ce situație ați lua deliberat această decizie?
7. Explicați mecanismul prin care o placă Arduino se resetează singură la începutul încărcării unui program și legați acest mecanism de faptul că deschiderea monitorului serial repornește programul.
8. Argumentați, cu numărul de cicluri de ceas, de ce `digitalWrite` este mult mai lent decât o scriere directă în registrul `PORTB`, și indicați o situație în care diferența contează și una în care nu contează.
9. Un program folosește douăzeci de mesaje constante de câte patruzeci de caractere, tipărite pe interfața serială. Estimați consumul de memorie SRAM și explicați cum îl reduce macroul `F()`.
10. Un program compilat cu `F_CPU` egal cu 8000000 rulează pe un microcontroler tactat efectiv la 16 MHz. Ce se întâmplă cu întârzierile și cu viteza interfeței seriale? Justificați prin calcul.
11. De ce este dificilă depanarea unui microcontroler comparativ cu depanarea unui program pe calculator? Enumerați cel puțin trei cauze de natură diferită.
12. Ce oferă modulul `debugWIRE`, care este dezavantajul activării siguranței `DWEN` pe un ATmega328P și de ce, în ciuda existenței lui, mesajele trimise pe interfața serială rămân metoda de depanare cea mai folosită?

4. Porturi de intrare și ieșire

Porturile de intrare și ieșire
Portul digital
Arhitectura porturilor
Configurarea porturilor
Transferul de date către microcontroler
Metastabilitatea
Circuitul sincronizator
Rezistențele de pull-up și de pull-down
Pinii cu funcții multiple
Protecția pinilor digitali
Circuitele conexe
Întrebări de verificare

Un microcontroler care nu poate atinge lumea din jurul lui nu este bun la nimic. Oricât de rapidă ar fi unitatea centrală și oricât de generoasă memoria, valoarea practică a sistemului se măsoară în ceea ce reușește să citească de la senzori și în ceea ce reușește să comande către actuatore. Legătura aceasta se face prin *porturile de intrare și ieșire*, iar acest capitol se ocupă de cea mai simplă și, tocmai de aceea, cea mai des folosită formă a lor: portul digital.

Simplitatea este însă înșelătoare. Un pin digital pare la prima vedere un fir pe care programul pune ori 0, ori 1. În realitate, în spatele fiecărui pin stă un mic circuit format din bistabile de configurare, tranzistoare de putere, diode de protecție, o rezistență comutabilă și un lanț de sincronizare cu ceasul intern. Fiecare dintre aceste elemente există dintr-un motiv precis, iar necunoașterea lui produce exact acele erori care par inexplicabile: un buton care își schimbă starea singur, o citire care întârzie cu un ciclu, un impuls scurt care nu se vede niciodată în program, un LED care arde portul. Capitolul urmărește, pe rând, structura hardware a unui pin, registrele software care îl controlează, drumul semnalului dinspre exterior spre nucleu, problemele care apar pe acest drum și circuitele exterioare care se conectează în mod obișnuit la porturi.

Microcontrolerul de referință este ATmega328P, cel de pe placa Arduino Uno. Familia AVR are avantajul unei organizări deosebit de clare a porturilor, cu trei registre pe port și cu o corespondență directă între bit și pin, ceea ce face ca discuția să rămână concretă fără să se piardă în particularități.

4.1 Porturile de intrare și ieșire

Porturile de intrare și ieșire, prescurtate curent I/O de la *input/output*, sunt interfețele hardware prin care microcontrolerul comunică cu perifericele externe. Prin ele trec semnalele de la butoane, comutatoare de capăt, senzori de proximitate, encodere sau alte circuite integrate, și tot prin ele pleacă spre exterior comenzile către LED-uri, relee, drivere de motor, afișoare sau module de comunicație. Un port nu conține prelucrare de semnal propriu-zisă; el este locul unde o valoare logică din memoria microcontrolerului devine tensiune pe un fir, sau invers.

Un pin este considerat pin de intrare și ieșire dacă sensul de transfer al datelor prin el poate fi schimbat prin program, dinspre intrare spre ieșire și înapoi, fără intervenție asupra circuitului fizic. Această reversibilitate prin software este trăsătura care deosebește un pin de I/O de un pin cu funcție fixă, cum ar fi pinul de alimentare sau pinul de reset. Ea este și motivul pentru care aceeași placă poate fi folosită într-o aplicație pentru a citi patru butoane, iar în alta pentru a comanda patru LED-uri, fără nicio modificare hardware.

Pinii nu sunt tratați individual de arhitectura internă, ci sunt grupați, de regulă câte opt, formând un *port*. Gruparea nu este arbitrară: opt biți înseamnă exact lățimea unui registru pe o mașină pe 8 biți, deci întregul port poate fi citit sau scris printr-o singură instrucțiune. Pe ATmega328P există trei porturi, numite B, C și D. Portul D are toți cei opt pini disponibili, portul B are opt pini din care doi sunt ocupați pe placa Arduino Uno de cuarțul de 16 MHz, iar portul C are șapte pini, dintre care ultimul este folosit ca intrare de reset. Se vede de aici o observație generală: numărul de pini scrise în foaia de catalog și numărul de pini efectiv utilizabili într-o placă concretă nu coincid, pentru că unele linii sunt deja angajate de circuitul de pe placă.

Există două tipuri de porturi de intrare și ieșire, după natura mărimii transferate. *Porturile digitale* lucrează cu două niveluri de tensiune, interpretate ca 1 logic și 0 logic; ele sunt potrivite pentru butoane, comutatoare, LED-uri, semnale de tact, adică pentru tot ce este prin natura lui binar. *Porturile analogice* măsoară o gamă continuă de tensiuni și sunt necesare atunci când informația este o mărime fizică variabilă, cum ar fi tensiunea de la un senzor de temperatură, de la un potențiometrul sau de la o punte tensometrică. Pe ATmega328P, porturile analogice nu sunt structuri separate, ci pini ai portului C care pot fi comutați spre convertorul analog-numeric intern; capitolul de față tratează numai funcționarea lor digitală.

Peste funcția de bază, mulți pini poartă și funcții speciale, puse la dispoziție de perifericele integrate: comunicație serială UART, SPI sau I2C, generare de semnal modulat în durată PWM, intrări de numărare pentru temporizatoare, intrări de întrerupere externă. Aceste funcții nu sunt gratuite: ele se activează prin registrele perifericului respectiv și, cât timp sunt active, confiscă pinul, care nu mai poate fi folosit ca pin obișnuit. Vom reveni asupra acestui conflict într-o secțiune dedicată.

4.2 Portul digital

Portul digital se sprijină pe o convenție simplă: orice tensiune aplicată pe pin este clasificată de microcontroler în una din două stări, 1 sau 0. Nu există „aproape 1” și nu există „pe jumătate”; circuitul de intrare este construit tocmai pentru a forța o decizie. Aceasta este marea comoditate a logicii digitale, dar și sursa primei categorii de erori, pentru că decizia se ia după praguri fixe, iar semnalul real nu ține cont de ele.

Intervalele de tensiune care corespund celor două stări depind de tipul microcontrolerului, de tensiunea de alimentare și de tehnologia de fabricație. Pentru un AVR alimentat la 5 V, foaia de catalog garantează că orice tensiune cuprinsă între 0 și $0,3 \cdot V_{CC}$, adică până la aproximativ 1,5 V, va fi citită sigur ca 0 logic, și că orice tensiune peste $0,6 \cdot V_{CC}$, adică peste aproximativ 3 V, va fi citită sigur ca 1 logic. Pentru un microcontroler cu nucleu ARM Cortex-M alimentat la 3,3 V, pragurile sunt raportate la aceeași tensiune de alimentare, dar cu alți coeficienți: sub $0,3 \cdot V_{CC}$ pentru 0 logic și peste circa $0,7 \cdot V_{CC}$ pentru 1 logic, adică sub 0,99 V, respectiv peste 2,31 V.

Se observă imediat că între cele două praguri rămâne o bandă intermediară, între 1,5 și 3 V în cazul AVR-ului la 5 V, pentru care foaia de catalog nu promite nimic. Nu înseamnă că microcontrolerul refuză să citească; înseamnă că rezultatul citirii nu este garantat și poate diferi de la exemplar la exemplar, de la temperatură la temperatură, sau chiar de la o citire la alta. Un proiectant prudent tratează această bandă ca pe o zonă interzisă și se asigură, prin construcția circuitului exterior, că semnalul o traversează repede și nu staționează în ea.

O consecință practică a raportării pragurilor la V_{CC} este că nivelurile logice nu sunt transferabile între sisteme alimentate diferit. Un microcontroler la 3,3 V care emite un 1 logic de 3,3 V se află, pentru un AVR alimentat la 5 V, peste pragul de 3 V, deci citirea funcționează; invers, un AVR care emite 5 V pe un pin de intrare al unui circuit la 3,3 V îi aplică acestuia o supratensiune și îl poate distruge. Interfațarea între domenii de tensiune diferite cere fie divizoare rezistive, fie translatoare de nivel dedicate.

În fine, corespondența dintre nivelul de tensiune și semnificația logică a semnalului este o alegere a proiectantului, nu o lege a naturii. În *logică pozitivă*, nivelul de tensiune ridicat înseamnă „adevărat” sau „activ”; în *logică negativă*, rolurile se inversează, iar starea activă este cea de tensiune joasă. Logica negativă apare foarte des în practică, tocmai pentru că montajul cu rezistență de pull-up, pe care îl vom studia mai departe, produce în mod natural 1 logic în repaus și 0 logic la apăsarea butonului. Confuzia dintre cele două convenții este una dintre cele mai frecvente cauze de comportament inversat în programele de început.

4.3 Arhitectura porturilor

Comunicația dintre microcontroler și lumea exterioară este bidirecțională, iar bidirecționalitatea trebuie construită undeva în hardware. Fiecare pin conține, în esență, trei elemente: un *buffer de ieșire*, care poate impune pe pin o tensiune, un *buffer de intrare*, care citește tensiunea de pe pin fără să o modifice, și o *linie de activare*, care hotărăște care dintre cele două este

conectat efectiv la pin. Linia de activare este cea care dă sensul transferului și ea este comandată direct de un bit dintr-un registru de configurare.

Când bufferul de ieșire este activ, pinul este condus de microcontroler: interiorul impune tensiunea, iar circuitul exterior o suportă. Când bufferul de ieșire este dezactivat, ieșirea trece în stare de înaltă impedanță și pinul devine, din punctul de vedere al circuitului exterior, aproape un fir întrerupt; tensiunea de pe el este acum determinată exclusiv de exterior, iar bufferul de intrare o observă. Este important de reținut că cele două bufele nu sunt niciodată active simultan pe același pin, pentru că un etaj de ieșire care ar încerca să impună 0 în timp ce exteriorul impune 5 V ar produce un scurtcircuit.

4.3.1 Structura internă a unui pin

Figura 4.1 arată, simplificat, ce se află în spatele unui singur pin de port pe un AVR. Se disting cinci blocuri, fiecare cu rolul lui.

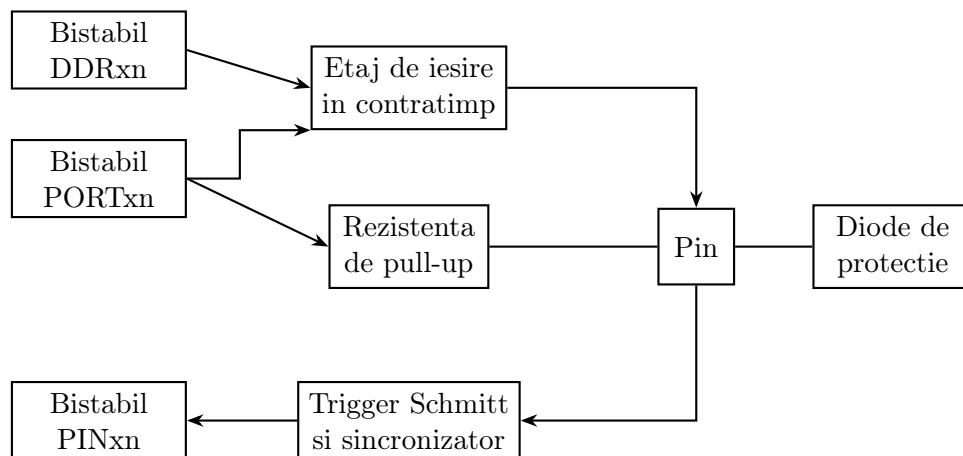


Figura 4.1 Structura simplificată a unui pin de port, cu bistabilele de configurare, etajul de ieșire, rezistența de pull-up comutabilă, diodele de protecție și lanțul de intrare.

Primul bloc este perechea de *bistabile de configurare*. Unul dintre ele memorează bitul de direcție și comandă activarea etajului de ieșire; celălalt memorează valoarea pe care programul dorește să o vadă pe pin atunci când acesta este ieșire. Aceste bistabile sunt de tip static: odată scrise de program, ele își păstrează starea până la o nouă scriere sau până la un reset, fără intervenția continuă a unității centrale. De aceea, un LED aprins printr-o singură instrucțiune rămâne aprins la nesfârșit, chiar dacă programul trece mai departe.

Al doilea bloc este *etajul de ieșire în contratimp*, cunoscut și sub numele englezesc de etaj push-pull. El este format din două tranzistoare complementare așezate în serie între alimentare și masă, cu pinul luat din punctul lor de mijloc. Când tranzistorul de sus conduce, pinul este legat la V_{CC} și poate furniza curent spre exterior, situație numită *source*. Când conduce cel de jos, pinul este legat la masă și poate absorbi curent din exterior, situație numită *sink*. Cele două tranzistoare nu conduc niciodată simultan, pentru că asta ar însemna un scurtcircuit între alimentare și masă prin interiorul cipului. Când etajul este dezactivat, ambele tranzistoare sunt blocate, iar pinul este lăsat în aer.

Al treilea bloc este *rezistența de pull-up comutabilă*, o rezistență de valoare mare, tipic între 20 și 50 k Ω la AVR, legată între V_{CC} și pin printr-un comutator electronic. Ea nu poate impune un nivel logic împotriva unui circuit exterior, pentru că prin ea trece un curent foarte mic, dar este perfect suficientă pentru a fixa potențialul unui pin care altfel ar rămâne nedefinit. Rolul ei complet este discutat în secțiunea dedicată.

Al patrulea bloc este format din *diodele de protecție*, două diode legate de la pin spre masă și de la pin spre alimentare, orientate astfel încât în funcționare normală să fie blocate. Dacă tensiunea de pe pin coboară sub masă sau urcă peste alimentare, dioda corespunzătoare intră în conducție și deviază curentul, limitând tensiunea pe structura internă la aproximativ o cădere de diodă în afara intervalului de alimentare. Este mecanismul care salvează cipul la descărcări electrostatice și la vârfuri de tensiune inductive de scurtă durată.

Al cincilea bloc este *lanțul de intrare*, format dintr-un circuit de tip trigger Schmitt și un sincronizator, urmat de bistabilul din care programul citește starea pinului. Acest lanț merită o discuție separată, pentru că el explică atât întârzierea la citire, cât și felul în care microcontrolerul se apără de metastabilitate.

4.3.2 Cele trei registre ale unui port

Din punct de vedere software, un port AVR este descris de trei registre, fiecare de 8 biți, cu bitul n corespunzând pinului n al portului. Pentru portul B acestea sunt `DDRB`, `PORTB` și `PINB`; pentru portul C și D, litera finală se schimbă corespunzător. Numele generic este `DDR $x$` , `PORT $x$` și `PIN $x$` .

Registrul `DDR $x$` , de la *Data Direction Register*, stabilește direcția fiecărui pin. Un bit șters, deci 0, configurează pinul respectiv ca intrare; un bit setat, deci 1, îl configurează ca ieșire. Fiind vorba de biți independenți, pe același port pot coexista fără nicio problemă pini de intrare și pini de ieșire, ceea ce se întâmplă de altfel în aproape orice aplicație reală. După reset, toți biții din `DDR $x$` sunt 0, adică toți pinii pornesc ca intrări; aceasta este alegerea sigură, pentru că o intrare nu poate deteriora nimic în circuitul exterior, în timp ce o ieșire configurată din greșeală peste un semnal extern poate produce un conflict.

Registrul `PORT $x$` are un dublu rol, iar înțelegerea acestui dublu rol este poate cel mai important lucru din tot capitolul. Pentru un pin configurat ca ieșire, `PORT $x$` conține valoarea logică pe care etajul de ieșire o impune pe pin: 1 înseamnă că pinul este tras la V_{CC} , 0 înseamnă că este tras la masă. Pentru un pin configurat ca intrare, etajul de ieșire este dezactivat și valoarea din `PORT $x$` nu mai ajunge pe pin; în schimb, ea comandă comutatorul rezistenței de pull-up. Un 1 scris în `PORT $x$` pentru un pin de intrare activează rezistența internă de pull-up, un 0 o dezactivează și lasă pinul în stare de înaltă impedanță.

Registrul `PIN $x$` este singurul care spune adevărul despre starea fizică a pinului. Citirea lui întoarce nivelul logic măsurat efectiv pe pin, indiferent de direcție și indiferent de ce a scris programul în `PORT $x$` . Pe AVR, `PIN $x$` nu este un registru obișnuit de memorie, ci fereastra spre bistabilul de la capătul lanțului de sincronizare; de aceea, o scriere în el nu se comportă ca o scriere normală. La ATmega328P, scrierea unui 1 într-un bit din `PIN $x$` basculează bitul corespunzător din `PORT $x$` , ceea ce oferă o metodă foarte rapidă, într-o singură instrucțiune,

de a inversa starea unui pin de ieșire.

De ce sunt necesare trei registre și nu unul singur? Pentru că cele trei informații pe care le poartă sunt logic independente și pot fi, în orice moment, diferite. Direcția este o proprietate de configurare, care se stabilește de obicei o singură dată, la inițializare. Valoarea de ieșire este o comandă a programului spre exterior. Starea citită este o informație venită dinspre exterior spre program. Un pin de ieșire configurat pe 1 poate fi măsurat, prin `PINx`, pe 0, dacă un scurtcircuit spre masă îl trage acolo; tocmai această discrepanță este utilă la diagnostic și ar fi imposibil de observat dacă registrul ar fi unul singur. Un pin de intrare cu pull-up activat are în `PORTx` valoarea 1, dar poate avea în `PINx` valoarea 0 dacă butonul este apăsat. A citi `PORTx` în loc de `PINx` pentru a afla starea unui buton este eroarea clasică a începătorului: programul va raporta imperturbabil 1, pentru că asta a scris el acolo.

Observație

Regula practică se ține minte în trei cuvinte: `DDRx` pentru direcție, `PORTx` pentru ce vrem noi, `PINx` pentru ce este de fapt. Orice citire a unei intrări se face din `PINx`. Citirea din `PORTx` întoarce doar ce a scris programul acolo, nu starea reală a pinului.

4.4 Configurarea porturilor

Unitatea centrală de procesare controlează porturile digitale numai după ce fiecare pin a fost configurat ca intrare sau ca ieșire. Configurarea este primul lucru care se face într-un program de microcontroler, înainte de bucla principală, și de corectitudinea ei depinde tot restul. Prin configurarea corectă a direcției se asigură nu numai funcționarea, ci și integritatea electrică a montajului: o direcție greșită poate pune față în față două surse de tensiune.

Combinațiile posibile ale celor doi biți de configurare, cel din `DDRx` și cel din `PORTx`, sunt patru și acoperă toate regimurile în care poate fi pus un pin. Tabelul 4.1 le rezumă.

Există și un mecanism global care poate anula linia a doua din tabel: bitul PUD din registrul `MCUCR`. Când acest bit este setat, toate rezistențele de pull-up ale tuturor porturilor sunt dezactivate simultan, indiferent de conținutul registrelor `PORTx`. Facilitatea este utilă în aplicațiile care urmăresc consumul minim, unde un pull-up activat pe un pin ținut la masă înseamnă un curent inutil, dar este o sursă serioasă de confuzie dacă bitul rămâne setat din greșală: montajele cu butoane par brusc să nu mai funcționeze, deși codul de inițializare pare corect.

4.4.1 Configurarea prin registre

Configurarea directă prin registre se face cu operații pe biți. Pentru a seta un bit fără a-i atinge pe ceilalți se folosește un SAU logic cu o mască având un singur bit pe 1; pentru a-l șterge se folosește un ȘI logic cu masca negată; pentru a-l bascula se folosește un SAU exclusiv. Masca se construiește prin deplasarea unui 1 pe poziția dorită.

Tabela 4.1 Regimul unui pin în funcție de biții din DDRx și PORTx.

DDR _{xn}	PORT _{xn}	Regim	Comportament electric
0	0	Intrare, înaltă impedanță	Pinul este lăsat în aer; nivelul citit este nedefinit dacă exteriorul nu îl impune.
0	1	Intrare cu pull-up	Rezistența internă trage pinul spre V_{CC} ; exteriorul îl poate coborî la masă printr-un contact.
1	0	Ieșire pe 0 logic	Pinul este legat la masă prin tranzistorul inferior și poate absorbi curent.
1	1	Ieșire pe 1 logic	Pinul este legat la V_{CC} prin tranzistorul superior și poate furniza curent.

```

1 #include <avr/io.h>
2
3 int main(void)
4 {
5     /* PB5 ca iesire: LED-ul de pe placa Arduino Uno */
6     DDRB |= (1 << PB5);
7
8     /* PD2 ca intrare cu rezistenta interna de pull-up activata */
9     DDRD &= ~(1 << PD2);
10    PORTD |= (1 << PD2);
11
12    while (1) {
13        /* butonul apasat leaga pinul la masa: se citeste 0 */
14        if ((PIND & (1 << PD2)) == 0) {
15            PORTB |= (1 << PB5);        /* aprinde LED-ul */
16        } else {
17            PORTB &= ~(1 << PB5);      /* stinge LED-ul */
18        }
19    }
20 }

```

Se observă că testul stării butonului se face pe registrul PIND, nu pe PORTD. Expresia $PIND \& (1 \ll PD2)$ izolează bitul dorit și dă fie zero, fie o valoare nenulă; comparația cu zero traduce apoi acest rezultat în condiția logică dorită. Este o eroare frecventă să se scrie $PIND == 0$, ceea ce ar cere ca toți cei opt pini ai portului să fie simultan pe zero.

Când mai mulți pini trebuie configurați odată, măștile se combină, iar codul devine mai compact fără să piardă din claritate.

Exemplu 4.2 Configurarea simultana a mai multor pini

```

1  /* PD3, PD4 si PD5 ca iesiri , restul portului neatins */
2  DDRD |= (1 << PD3) | (1 << PD4) | (1 << PD5);
3
4  /* toate cele trei iesiri pe zero */
5  PORTD &= ~( (1 << PD3) | (1 << PD4) | (1 << PD5) );
6
7  /* basculeaza PD3 fara a citi mai intai valoarea curenta */
8  PIND = (1 << PD3);

```

Ultima linie folosește particularitatea amintită mai devreme: scrierea unui 1 în PINx basculează bitul corespunzător din PORTx. Efectul este identic cu `PORTD ^= (1 << PD3)`, dar se obține fără ciclul de citire-modificare-scriere.

4.4.2 Configurarea prin funcțiile Arduino

Mediul Arduino ascunde registrele în spatele a trei funcții: `pinMode` pentru direcție și pull-up, `digitalWrite` pentru valoarea de ieșire și `digitalRead` pentru citire. Programul echivalent celui de mai sus devine considerabil mai scurt.

Exemplu 4.3 Acelasi montaj scris cu functiile Arduino

```

1  const int LED      = 13;    /* corespunde pinului PB5 */
2  const int BUTON    = 2;     /* corespunde pinului PD2 */
3
4  void setup()
5  {
6      pinMode(LED, OUTPUT);
7      pinMode(BUTON, INPUT_PULLUP);
8  }
9
10 void loop()
11 {
12     if (digitalRead(BUTON) == LOW) {
13         digitalWrite(LED, HIGH);
14     } else {
15         digitalWrite(LED, LOW);
16     }
17 }

```

Modul `INPUT_PULLUP` face, într-un singur apel, exact ce făceau cele două linii cu `DDRD` și `PORTD`: șterge bitul de direcție și setează bitul din `PORTx`. Modul `INPUT` simplu lasă pinul în

înalță impedanță, fără pull-up, ceea ce este corect doar dacă nivelul este impus din exterior.

Diferența dintre cele două stiluri nu este doar de gust, ci și de cost. O scriere de forma `PORTB |= (1 << PB5)` se compilează, pentru un bit dintr-un registru aflat în zona de adrese joase, într-o singură instrucțiune mașină, executată într-un ciclu de ceas. Un apel `digitalWrite` face mult mai multe: traduce numărul pinului Arduino în port și bit prin două tabele aflate în memoria de program, verifică dacă pe pinul respectiv este activ un canal PWM și îl oprește dacă este cazul, dezactivează întreruperile pe durata operației atomice de citire-modificare-scriere, apoi le restaurează. Costul total este de ordinul zecilor de cicluri de ceas, adică de câteva microsecunde la 16 MHz. Pentru aprins un LED comandat de un buton, diferența este complet irelevantă. Pentru generarea unui semnal cu front precis, pentru un protocol implementat software sau pentru o rutină de tratare a întreruperii care trebuie să se termine repede, diferența devine hotărâtoare.

Observație

Regula de bun-simț este să se scrie codul cu funcțiile Arduino atunci când claritatea contează mai mult decât viteza, și direct cu registre acolo unde se numără ciclii. Cele două stiluri se pot amesteca în același program, pentru că funcțiile Arduino nu fac altceva decât să scrie în aceleași registre.

4.5 Transferul de date către microcontroler

Până acum am privit pinul din afară. Rămâne de urmărit ce se întâmplă cu tensiunea aplicată pe pin din momentul în care ea există fizic până în momentul în care programul o poate testa. Drumul acesta nu este instantaneu și nu este nici măcar continuu: el este tăiat în felii de ceasul sistemului.

Transformarea tensiunii de intrare într-o valoare binară presupune, pe lângă tranzistoarele din etajul de intrare, cel puțin un circuit suplimentar de memorare. Soluția uzuală este folosirea unui bistabil de tip *latch*, adică a unui element care preia valoarea de la intrare într-un anumit moment și o păstrează până la momentul următor. Dacă acest bistabil este comandat de ceasul sistemului, atunci reținerea valorii de pe pin se face o singură dată pe ciclu, la un front bine precizat al ceasului.

Consecința imediată este că starea pinului nu este citită continuu, ci eșantionată. Programul nu vede pinul, ci vede ultima fotografie făcută pinului. Dacă tensiunea se schimbă imediat după momentul de eșantionare, schimbarea va fi vizibilă abia la eșantionarea următoare, deci cu o întârziere de până la un ciclu întreg de ceas. La AVR, foaia de catalog spune că întârzierea între schimbarea fizică a pinului și disponibilitatea noii valori în registrul `PINx` este cuprinsă între jumătate și un ciclu și jumătate de ceas, în funcție de momentul în care se produce schimbarea față de front. La 16 MHz, un ciclu înseamnă 62,5 ns, deci întârzierea este de ordinul zecilor de nanosecunde, neglijabilă pentru butoane, dar deloc neglijabilă pentru un protocol sincron rapid.

A doua consecință, mai supărătoare, este că impulsurile mai scurte decât un ciclu de ceas pot fi pierdute complet. Dacă semnalul urcă și coboară între două momente de eșantionare,

bistabilul nu are cum să afle că s-a întâmplat ceva: la ambele eșantionări valoarea era aceeași. Este ca și cum am fotografia o ușă o dată pe secundă și am spera să prindem pe cineva care intră și iese într-o zecime de secundă. Din acest motiv, detectarea impulsurilor foarte scurte nu se face prin citirea repetată a lui $PINx$ din program, ci prin mecanisme hardware dedicate: întreruperile externe declanșate pe front, care au propriul circuit de detecție, sau captura pe temporizator, care memorează chiar valoarea numărătorului în momentul frontului.

Exemplu

Un encoder incremental montat pe un ax produce impulsuri cu durată de 200 ns la turația maximă. Poate fi citit prin interogarea registrului $PINx$ într-o buclă, pe un ATmega328P la 16 MHz?

Rezolvare. Un ciclu de ceas durează $1/16 \text{ MHz} = 62,5 \text{ ns}$, deci impulsul acoperă aproximativ trei cicluri. Eșantionarea propriu-zisă a pinului l-ar putea prinde. Problema este însă bucla de program: o iterație care citește $PINx$, maschează bitul, compară și face un salt condiționat consumă cel puțin cinci-șase cicluri, adică peste 350 ns, mai mult decât durată impulsului. Există deci iterații în care impulsul se naște și moare între două citiri consecutive, iar el se pierde. Soluția corectă este folosirea unei întreruperi externe pe front, care nu depinde de momentul în care programul apucă să citească.

4.6 Metastabilitatea

Eșantionarea ridică o problemă mai subtilă decât întârzierea, și anume ce se întâmplă atunci când semnalul de intrare se schimbă chiar în momentul în care este eșantionat. Fenomenul poartă numele de *metastabilitate* și înseamnă o stare instabilă în care ieșirea unui bistabil se află într-o zonă de tranziție între cele două niveluri logice, fără a fi nici 0, nici 1, pentru o durată care nu poate fi prezisă.

Ca să înțelegem de ce se poate întâmpla așa ceva, trebuie privit înăuntrul bistabilului. Un bistabil își păstrează valoarea printr-o buclă de reacție pozitivă: două inversoare legate în cerc, ieșirea fiecăruia comandând intrarea celuilalt. Bucla are două stări stabile, corespunzătoare celor două valori logice, dar are și o a treia stare, de echilibru instabil, în care ambele inversoare se află la mijlocul caracteristicii lor și se alimentează reciproc cu exact aceeași tensiune intermediară. Este situația bilei așezate exact în vârful unui deal: în principiu poate sta acolo, în practică va cădea, dar nu se știe dinainte nici încotro, nici după cât timp.

Bistabilul ajunge în această stare atunci când, în momentul frontului de ceas, tensiunea de la intrarea lui nu este destul de departe de pragul de comutare. Foile de catalog exprimă condiția prin doi timpi: *timpul de stabilizare* dinaintea frontului, în engleză *setup time*, și *timpul de menținere* de după front, *hold time*. Dacă semnalul este stabil pe tot intervalul format din cei doi timpi, bistabilul comută curat. Dacă semnalul se schimbă chiar în această fereastră, energia injectată în buclă nu este suficientă pentru a o împinge decis spre una din stările stabile și bucla rămâne un timp nedecisă.

Ieșirea din starea metastabilă se face exponențial: probabilitatea ca bistabilul să fie încă

nedecis scade exponențial cu timpul care trece de la front. Nu există un timp după care ieșirea din metastabilitate este garantată matematic, dar există un timp după care probabilitatea de a mai fi nedecis devine atât de mică încât evenimentul nu se produce practic niciodată pe durata de viață a echipamentului. Această observație stă la baza tuturor soluțiilor practice: metastabilitatea nu se elimină, se face improbabilă.

Cauzele care favorizează apariția ei sunt de două feluri. Prima, cea fundamentală, este lipsa de corelație între semnalul extern și ceasul microcontrolerului. Un buton apăsat de un om, un semnal venit de la un alt circuit cu propriul cuarț sau ieșirea unui senzor sunt *asincrone* față de ceasul intern; nimic nu le împiedică să comute exact în fereastra critică, și, dacă sistemul funcționează suficient de mult timp, mai devreme sau mai târziu asta se va întâmpla. A doua cauză, tehnologică, ține de calitatea fronturilor: un semnal care parcurge un cablu lung ajunge la destinație cu fronturi lente, deformate de capacitatea și inductanța liniei și eventual însoțite de reflexii. Un front lent petrece mult timp în zona intermediară de tensiune, deci mărește direct probabilitatea ca eșantionarea să îl prindă acolo.

Împotriva fronturilor lente, remediul este circuitul de tip *trigger Schmitt*, prezent la intrarea fiecărui pin de AVR. Un comparator obișnuit are un singur prag și, dacă semnalul de intrare traversează lent acest prag purtând și puțin zgomot, ieșirea lui oscilează de mai multe ori între 0 și 1 la fiecare tranziție. Triggerul Schmitt are două praguri, unul superior V_{hi} pentru trecerea din 0 în 1 și unul inferior V_{lo} pentru trecerea din 1 în 0. După ce a comutat în sus, circuitul își coboară singur pragul de decizie, astfel încât zgomotul suprapus peste semnal nu mai este suficient pentru a-l face să comute înapoi. Diferența dintre praguri se numește *histerezis* și este o măsură a imunității la zgomot. Rezultatul este că, indiferent cât de lent urcă tensiunea la intrare, ieșirea triggerului face o singură tranziție, curată și rapidă.

Trebuie însă spus limpede că triggerul Schmitt nu rezolvă metastabilitatea, ci doar una dintre cauzele ei. El curăță frontul și îl face abrupt, reducând mult timpul petrecut în zona nedefinită, dar nu poate face nimic în privința momentului în care acest front se produce față de ceas. Un front perfect abrupt care cade exact în fereastra de setup-hold produce metastabilitate la fel de bine ca unul lent. Pentru momentul frontului este nevoie de altceva.

4.7 Circuitul sincronizator

Soluția pentru caracterul asincron al semnalelor externe este *sincronizatorul*, un circuit care nu elimină metastabilitatea, ci îi acordă timp să se stingă înainte ca valoarea să ajungă la program. Ideea este simplă și eficientă: se pun două bistabile în serie, comandate de același ceas, iar valoarea folosită mai departe este cea de la ieșirea celui de-al doilea.

Mecanismul se înțelege urmărind Figura 4.2. Primul bistabil este cel expus riscului: el eșantionează un semnal asincron și poate intra în metastabilitate. Ieșirea lui nu merge însă direct spre program, ci spre al doilea bistabil, care o va eșantiona abia la ciclul următor de ceas. În intervalul de un ciclu întreg dintre cele două fronturi, primul bistabil are timp să se hotărască. Probabilitatea ca el să fie încă nedecis după un ciclu complet este, așa cum am văzut, exponențial mică, deci al doilea bistabil eșantionează, practic întotdeauna, un semnal

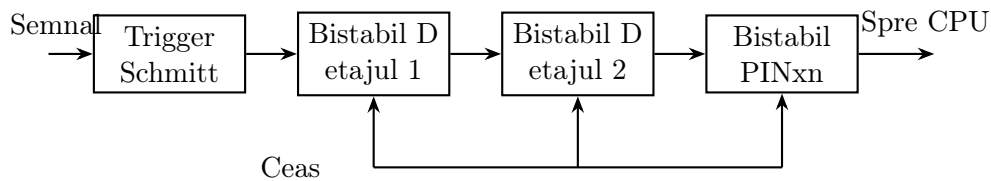


Figura 4.2 Lanțul de intrare al unui pin: trigger Schmitt, sincronizator cu două etaje și bistabilul din care se citește registrul PINx.

deja curat, și trece mai departe o valoare logică bine definită.

Ce nu face sincronizatorul este să garanteze care dintre cele două valori va fi. Dacă schimbarea a picat exact pe front, valoarea propagată poate fi cea veche sau cea nouă, iar care anume nu se poate ști. Din punctul de vedere al programului, asta înseamnă doar că tranziția a fost văzută cu un ciclu mai devreme sau cu un ciclu mai târziu, ceea ce este aproape întotdeauna acceptabil. Ce se garantează este că programul nu va vedea niciodată o valoare care nu este nici 0, nici 1, și, mai important, că două părți diferite ale circuitului nu vor interpreta diferit aceeași citire nedecisă.

Se poate defini astfel o mărime d_{sinc} , numărul de cicluri de ceas necesare propagării semnalului de la primul bistabil de sincronizare până la bistabilul de citire al pinului, și o mărime d_{latch} , întârzierea introdusă de memorarea propriu-zisă. Întârzierea totală la intrare este suma lor:

$$d_{in} = d_{latch} + d_{sinc}.$$

Cu cât sincronizatorul are mai multe etaje, cu atât probabilitatea de eroare scade, dar cu atât crește și d_{sinc} . Două etaje reprezintă compromisul obișnuit pentru frecvențele de lucru ale microcontrolerelor uzuale; sistemele foarte rapide, la sute de megahertzi, sau cele cu cerințe stricte de siguranță pot folosi trei sau mai multe.

Observație

Sincronizatorul se aplică semnalului, nu magistralei. Dacă același semnal asincron trebuie folosit în două locuri diferite din circuit, el trebuie sincronizat o singură dată, iar rezultatul distribuit; două sincronizatoare paralele pe același semnal pot da, în cazul nefericit, rezultate diferite, iar circuitul ajunge într-o stare inconsistentă.

4.8 Rezistențele de pull-up și de pull-down

O intrare digitală care nu este conectată la nicio sursă de semnal definită se află în stare nedeterminată sau, în termeni de circuit, în *înaltă impedanță*, prescurtat HiZ. Este poate cea mai contraintuitivă situație pentru un începător, pentru că simțul comun spune că un fir nelegat „este la zero”. Nu este.

Motivul este că intrarea unui port CMOS are o rezistență de intrare enormă, de ordinul sutelor de megaohmi, și un curent de scurgere de ordinul zecimilor de microamperi. Un pin lăsat în aer se comportă ca armătura unei capacități minuscule și izolate, al cărei potențial

este stabilit de orice altceva se întâmplă în jur: cuplajul capacitiv cu traseele vecine de pe placă, câmpul electromagnetic al rețelei de 50 Hz, sarcina electrostatică a mâinii care se apropie de placă. Tensiunea rezultată este de obicei undeva prin zona intermediară dintre praguri, iar citirile pe un asemenea pin par să se schimbe singure, în mod aparent aleatoriu. Un experiment clasic constă în a citi un pin nelegat și a afișa valoarea: se obține un șir care se modifică atunci când se apropie mâna de placă.

Există și un al doilea inconvenient, mai puțin vizibil dar la fel de real. Când tensiunea de pe pin staționează în zona intermediară, ambele tranzistoare din etajul de intrare conduc parțial și prin ele circulă un curent care nu are niciun rol util. Într-o aplicație alimentată de la baterie, câțiva pini lăsați în aer pot dubla consumul în repaus. Recomandarea generală, la orice montaj care urmărește consumul, este ca niciun pin să nu rămână flotant.

Trebuie totuși spus că starea de înaltă impedanță nu este întotdeauna un defect. Ea este, în anumite aplicații, exact ceea ce se dorește: un pin în HiZ nu influențează circuitul și nici nu este influențat de el, fiind izolat din punct de vedere electric. Pe această proprietate se bazează magistralele partajate, unde mai multe dispozitive sunt legate la aceeași linie și doar unul dintre ele o conduce la un moment dat, celelalte stând în HiZ. Diferența dintre folosire corectă și defect este intenția: HiZ este util când cineva anume răspunde de nivelul liniei și dăunător când nu răspunde nimeni.

4.8.1 Cum funcționează un pull-up

Rezolvarea situației în care nimeni nu impune nivelul este *rezistența de pull-up*, o rezistență legată între linia de semnal și tensiunea de alimentare. Ea injectează în permanență un curent mic spre pin; cum intrarea nu consumă practic nimic, potențialul pinului urcă la V_{CC} și starea în repaus devine 1 logic, bine definit. Când un element exterior, de exemplu un buton, leagă pinul la masă, el impune 0 logic fără efort, iar prin rezistență se stabilește un curent mic dinspre alimentare spre masă. Simetric, *rezistența de pull-down* se leagă între linie și masă și fixează starea de repaus la 0 logic, urmând ca exteriorul să o poată ridica la V_{CC} .

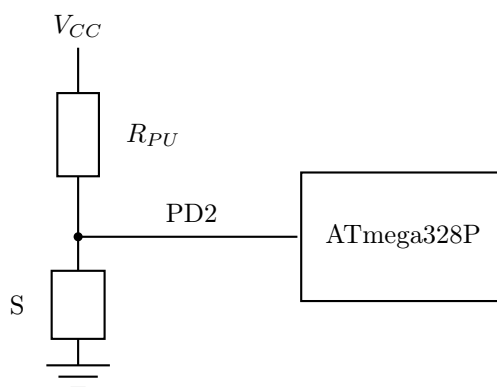


Figura 4.3 Buton cu rezistență de pull-up. În repaus pinul citește 1 logic; cu butonul apăsat citește 0 logic.

Figura 4.3 arată montajul clasic al unui buton cu pull-up, care produce logica nega-

tivă despre care am vorbit la începutul capitolului: repaus înseamnă 1, apăsăat înseamnă 0. Montajul complementar, cu pull-down și cu butonul legat la V_{CC} , produce logică pozitivă și pare mai natural, dar este mai rar folosit din două motive practice: majoritatea microcontrolerelor au pull-up intern și nu au pull-down intern, iar traseul de masă este de obicei mai bine distribuit pe placă decât cel de alimentare, deci contactul spre masă este mai ușor de realizat.

Valorile uzuale ale rezistențelor de pull-up și pull-down externe se situează între 1 și 10 k Ω , alegerea concretă fiind un compromis. O rezistență mică fixează ferm nivelul și încarcă rapid capacitatea parazită a liniei, deci este imună la zgomot și rapidă, dar consumă mai mult curent când linia este trasă la masă. O rezistență mare consumă puțin, dar lasă linia moale, sensibilă la interferențe, și încetinește fronturile, pentru că trebuie să încarce capacitatea traseului printr-un curent mic. Pentru un buton obișnuit, 10 k Ω este alegerea implicită; pentru o magistrală I2C rapidă sau pentru o linie lungă, valorile coboară spre 2,2 sau 1 k Ω .

Exemplu

Un buton este legat conform Figurii 4.3, cu $R_{PU} = 10$ k Ω , la o alimentare de 5 V. Ce curent circulă prin rezistență cu butonul apăsăat și cât consumă montajul dacă butonul este ținut apăsăat permanent?

Rezolvare. Cu butonul apăsăat, rezistența este conectată direct între 5 V și masă, căderea pe contactul închis fiind neglijabilă. Rezultă $I = 5 \text{ V} / 10 \text{ k}\Omega = 0,5 \text{ mA}$. Puterea disipată pe rezistență este $P = U \cdot I = 5 \cdot 0,5 \cdot 10^{-3} = 2,5 \text{ mW}$, adică nimic pentru o rezistență obișnuită de 0,25 W. Dacă în loc de 10 k Ω s-ar folosi 100 Ω , curentul ar deveni 50 mA și puterea 250 mW, ceea ce ar fi o risipă inacceptabilă într-un montaj alimentat de la baterie și ar încălzi vizibil rezistența. Cu pull-up-ul intern al AVR-ului, de 20 până la 50 k Ω , curentul ar fi între 0,1 și 0,25 mA.

4.8.2 Pull-up-ul intern al AVR-ului și când nu este suficient

Am văzut deja mecanismul software: pentru un pin configurat ca intrare, scrierea unui 1 în bitul corespunzător din PORTx conectează rezistența internă între pin și V_{CC} . Rezistența nu este precisă, valoarea ei fiind garantată doar într-un interval larg, tipic între 20 și 50 k Ω , pentru că este realizată dintr-un tranzistor MOS polarizat, nu dintr-un element de precizie. Pentru scopul ei, precizia nu contează: nu se cere decât să tragă spre alimentare un pin care nu consumă nimic.

Un aspect important este că rezistența de pull-up internă nu afectează în niciun fel pinii configurați ca ieșire. Pentru aceștia, bitul din PORTx comandă etajul de ieșire, iar comutatorul de pull-up este dezactivat de logica internă. Pull-up-ul îi influențează deci doar pe pinii de intrare, și îi influențează exclusiv în bine, dându-le o stare de repaus definită.

Există totuși situații în care pull-up-ul intern nu este suficient și trebuie completat sau înlocuit cu unul extern. Prima este aceea a liniilor lungi sau a mediilor cu perturbații electromagnetice puternice, de exemplu în vecinătatea unui motor comandat prin PWM: o

rezistență de zeci de kilohmi lasă linia prea moale, iar o perturbație capacitivă poate depăși pragul de comutare. Un pull-up extern de 1 sau 2,2 k Ω scade impedanța liniei și restabilește imunitatea. A doua situație este cea a magistralelor cu ieșiri în colector deschis, în primul rând I2C, unde rezistența de pull-up nu are doar rolul de a defini nivelul, ci și acela de a încărca linia suficient de rapid pentru a respecta timpii de front impuși de protocol; valorile interne sunt aici mult prea mari. A treia situație apare atunci când nivelul de repaus trebuie să existe și înainte ca microcontrolerul să apuce să își execute codul de inițializare, sau chiar cu microcontrolerul în reset: un semnal de validare a unui driver de motor trebuie ținut în starea inactivă de o rezistență externă, altfel motorul poate porni necontrolat în intervalul dintre alimentare și inițializare. În sfârșit, un pull-down este întotdeauna extern pe AVR, pentru că microcontrolerul nu dispune de rezistențe interne de acest tip.

4.9 Pinii cu funcții multiple

Numărul de picioare ale unei capsule este limitat de dimensiunea fizică și de costul încapsulării, în timp ce numărul de periferice pe care le poate conține un cip modern crește continuu. Rezolvarea acestei tensiuni este multiplexarea funcțiilor: același pin poate fi folosit, în momente diferite, de periferice diferite. Când este folosit în modul obișnuit, ca simplu pin digital, el se numește *GPIO*, de la *general purpose input/output*.

Funcțiile alternative uzuale acoperă întreaga gamă de periferice ale microcontrolerului. Intrarea analogică leagă pinul la multiplexorul convertorului analog-numeric, permițând măsurarea unei tensiuni. Liniile de comunicație serială scot pinul din controlul portului și îl dau modulului UART, SPI sau I2C, care îl comandă după protocolul propriu. Intrările de întrerupere externă atașează pinului un circuit de detecție a fronturilor, care poate întrerupe programul la o schimbare de stare. Ieșirile de comparare ale temporizatoarelor permit generarea de semnale PWM, adică forme de undă dreptunghiulare cu factor de umplere programabil, folosite pentru reglarea turăției motoarelor sau a intensității luminoase.

Configurarea acestor funcții nu se face din registrele portului, ci din *registrele de funcții speciale*, prescurtat SFR, ale perifericului respectiv. Pe ATmega328P, de exemplu, activarea emițătorului UART se face prin biți din registrul UCSR0B, activarea modulului SPI prin bitul SPE din SPCR, iar generarea PWM prin biții de mod din registrele TCCR. În toate cazurile, în momentul în care perifericul este activat, o poartă internă comută sursa care comandă pinul de la logica portului la logica perifericului.

Consecința este categorică: cât timp o funcție alternativă este activă, pinul nu mai poate funcționa ca pin cu scop general. Scrierile în PORTx pentru acel bit nu mai au efect asupra tensiunii de pe pin, iar citirile din PINx pot să nu mai reflecte ceea ce se așteaptă. Aceasta este sursa unei întregi familii de erori greu de diagnosticat, pentru că programul pare corect, iar pinul pur și simplu nu răspunde. Pe placa Arduino Uno, cazul cel mai frecvent este acela al pinilor 0 și 1, care corespund liniilor RXD și TXD ale portului serial: orice apel la `Serial.begin` confiscă acești doi pini, iar dacă la ei este legat și ceva din montaj, comunicația cu calculatorul și montajul se deranjează reciproc.

Un al doilea caz clasic, tot pe Arduino Uno, privește interacțiunea dintre funcțiile `tone`

și `analogWrite`, care folosesc același temporizator, sau folosirea bibliotecii `Servo`, care preia temporizatorul 1 și dezactivează astfel ieșirile PWM de pe pinii 9 și 10. Regula de lucru care evită aceste conflicte este simplă: înainte de a atribui o funcție unui pin, se verifică în tabelul de funcții alternative al microcontrolerului ce alte periferice îl folosesc, iar planificarea alocării pinilor se face la începutul proiectului, nu pe măsură ce se adaugă module.

Tabela 4.2 Câteva funcții alternative pe pinii ATmega328P și efectul activării lor.

Pin	Funcție alternativă	Efect asupra utilizării ca GPIO
PD0, PD1	RXD, TXD (UART)	Pinii sunt preluați de modulul serial; pe Arduino Uno sunt legați și la interfața USB.
PB2 la PB5	SS, MOSI, MISO, SCK (SPI)	Activarea SPI preia pinii; SCK este și pinul LED-ului de pe placă.
PC4, PC5	SDA, SCL (I2C)	Preluați de modulul TWI, cu pull-up extern obligatoriu.
PD2, PD3	INT0, INT1	Rămân citibili, dar generează întreruperi la front dacă sunt activate.
PC0 la PC5	Intrări ADC	Funcția digitală poate fi dezactivată din DIDR0 pentru a reduce consumul.

4.10 Protecția pinilor digitali

Pinii unui microcontroler sunt puncte de contact între o structură de siliciu de câțiva nanometri grosime și o lume exterioară plină de tensiuni parazite, sarcini electrostatice și sarcini inductive. Producătorii prevăd de aceea o serie de protecții interne, dar acestea au limite bine definite, iar depășirea lor distruge cipul. Cunoașterea limitelor este partea practică cea mai importantă a acestui capitol.

4.10.1 Protecțiile interne

Protecția la *descărcări electrostatice*, prescurtat ESD, este realizată prin diodele de la fiecare pin spre alimentare și spre masă, despre care am vorbit la structura internă a pinului. O descărcare electrostatică poate atinge mii de volți, dar transportă o energie foarte mică; diodele deviază impulsul de curent, iar tensiunea pe structura sensibilă rămâne limitată. Protecția este proiectată pentru evenimente scurte și rare, nu pentru o supratensiune permanentă.

Protecția la supratensiune se realizează prin aceleași diode, care lucrează aici în regim de *circuit de limitare*, în engleză *clamping*. Dacă tensiunea pe pin depășește V_{CC} cu o cădere de diodă, dioda superioară intră în conducție și scurge excesul spre alimentare. Trebuie subliniat că această protecție funcționează numai dacă prin diodă circulă un curent mic, de ordinul miliamperilor. Foaia de catalog a AVR-ului precizează o valoare maximă a curentului

prin dioda de limitare, iar depășirea ei nu numai că distruge dioda, dar poate ridica întreaga alimentare a cipului, cu efecte imprevizibile asupra celorlalte circuite. De aceea, la conectarea unui semnal care poate depăși alimentarea se pune întotdeauna o rezistență în serie, care limitează curentul prin diodă.

Protecția la supracurent și la scurtcircuit este prezentă în multe microcontrolere moderne sub forma unor limitatoare care reduc curentul prin pin când detectează o valoare excesivă. La AVR-urile clasice, această protecție nu există: etajul de ieșire nu are limitare proprie, iar un scurtcircuit între un pin de ieșire pe 1 și masă produce un curent limitat doar de rezistența internă a tranzistorului, de ordinul zecilor de miliamperi. Rezultatul tipic nu este distrugerea imediată, ci degradarea progresivă a etajului: pinul începe să comande „mai slab”, apoi rămâne blocat pe o valoare, iar în cele din urmă întreaga alimentare a cipului poate fi afectată.

Protecția la inversarea polarității alimentării nu este, în general, disponibilă pe cip și se implementează la nivel de placă, printr-o diodă în serie cu sursa de alimentare sau, mai eficient, printr-un tranzistor MOS montat în calea de retur. Protecția termică, cu senzor intern care reduce frecvența sau oprește circuitul la depășirea unui prag, există la microcontrolerele mai complexe; pe AVR-urile mici, senzorul intern de temperatură există, dar nu este legat de un mecanism automat de protecție.

4.10.2 Curentul maxim pe pin și pe port

Limitele de curent pe care le impune foaia de catalog sunt de trei feluri, iar toate trei trebuie respectate simultan.

Prima este limita pe pin. Pentru ATmega328P, valoarea maximă absolută a curentului pe un singur pin este de 40 mA, indiferent de sens. Aceasta este o valoare de distrugere, nu una de proiectare: la 40 mA pinul funcționează, dar tensiunea de ieșire se apropie periculos de pragul logic, disipația crește și fiabilitatea pe termen lung scade. Valoarea recomandată pentru proiectare este de aproximativ 20 mA, iar pentru montaje care trebuie să dureze ani de zile este rezonabil să se coboare la 10 mA.

A doua este limita pe grup de pini. Pinii sunt legați în interiorul capsulei la piste de alimentare comune, iar aceste piste au propria lor capacitate de transport. Foaia de catalog precizează pentru ATmega328P o limită de 100 mA pentru suma curenților dintr-un grup de pini, grupurile fiind definite explicit în documentație. Consecința practică este că nu se pot aprinde, de exemplu, opt LED-uri la 20 mA fiecare pe același grup, chiar dacă fiecare pin în parte respectă limita individuală.

A treia este limita totală, dată de curentul care poate trece prin picioarele de alimentare și de masă ale capsulei. Pentru ATmega328P ea este de 200 mA și include tot: consumul propriu al nucleului, al perifericelor și tot ce trece prin pinii de ieșire.

Regula de proiectare care decurge de aici este ca microcontrolerul să nu fie folosit ca sursă de putere. El este un element de comandă, iar tot ce are nevoie de mai mult de câțiva miliamperi, adică relee, motoare, benzi de LED-uri, electrovalve, se comandă printr-un element intermediar: tranzistor, driver integrat, releu sau optocuplor.

4.10.3 Rezistența de limitare pentru un LED

Cazul cel mai des întâlnit de sarcină legată direct la un pin este LED-ul, iar el ilustrează perfect de ce este nevoie de o rezistență în serie. O diodă luminescentă nu se comportă ca o rezistență: curentul prin ea crește exponențial cu tensiunea aplicată, iar tensiunea la bornele ei rămâne aproximativ constantă odată ce dioda s-a deschis. Această tensiune, numită *tensiune directă* și notată V_F , depinde de culoare, adică de materialul semiconductor: aproximativ 1,8 până la 2,2 V pentru roșu, în jur de 3 V pentru albastru și alb.

Dacă un LED roșu s-ar lega direct între un pin pe 5 V și masă, diferența dintre cei 5 V ai pinului și cei aproximativ 2 V ai LED-ului ar trebui să cadă undeva. Nu are unde: singurele rezistențe din circuit sunt rezistența internă a etajului de ieșire și rezistența dinamică foarte mică a LED-ului. Curentul crește necontrolat până la o valoare care distruge fie LED-ul, fie pinul, fie pe amândouă. Rolul rezistenței în serie este tocmai să preia diferența de tensiune și să fixeze curentul.

Valoarea rezistenței se calculează direct din legea lui Ohm aplicată pe rezistență:

$$R = \frac{V_{CC} - V_F}{I_F},$$

unde I_F este curentul dorit prin LED. Puterea disipată pe rezistență este $P = (V_{CC} - V_F) \cdot I_F$ și trebuie comparată cu puterea nominală a componentei.

Exemplu

Se cere comanda unui LED roșu cu $V_F = 2$ V de la un pin al unui ATmega328P alimentat la 5 V, la un curent de 10 mA. Să se calculeze rezistența de limitare, puterea disipată pe ea și numărul maxim de asemenea LED-uri care pot fi aprinse simultan pe un grup de pini.

Rezolvare. Tensiunea care trebuie să cadă pe rezistență este $5 - 2 = 3$ V. Rezultă

$$R = \frac{3 \text{ V}}{10 \text{ mA}} = 300 \, \Omega.$$

Valoarea standard imediat superioară din seria E12 este 330 Ω , care este și alegerea uzuală; ea dă un curent de $3/330 \approx 9$ mA, perfect acceptabil. Puterea disipată pe rezistență este $P = 3 \cdot 9 \cdot 10^{-3} \approx 27$ mW, deci o rezistență obișnuită de 0,25 W este mult mai mult decât suficientă.

Pentru numărul de LED-uri, limita de grup este de 100 mA. La 9 mA pe LED rezultă $100/9 \approx 11$ LED-uri, dar acest calcul consumă întreaga rezervă a grupului și nu lasă nimic pentru celelalte sarcini. O proiectare prudentă se oprește la jumătate, adică la cinci sau șase LED-uri simultan pe grup. Dacă aplicația cere mai multe, soluțiile sunt multiplexarea în timp, care reduce curentul mediu, sau folosirea unui driver extern.

Trebuie observat că, dacă în loc de 330 Ω s-ar monta 100 Ω , curentul ar deveni $3/100 = 30$ mA, sub maximul absolut de 40 mA, deci LED-ul ar lumina și circuitul ar părea să

funcționeze. Pinul ar lucra însă permanent în zona nerecomandată, iar trei asemenea LED-uri ar depăși deja limita de grup.

4.11 Circuitele conexe

Porturile digitale se conectează rareori direct la ceea ce trebuie citit sau comandat. Între ele se interpun circuite care adaptează nivelurile, izolează, filtrează sau amplifică. Această ultimă secțiune trece în revistă cele mai frecvente dintre ele.

4.11.1 Butoane și comutatoare

Un buton mecanic pare cel mai simplu element imaginabil, dar el produce în mod obișnuit un fenomen numit *oscilație de contact*, în engleză *bouncing*. Contactele metalice nu se lipesc instantaneu și definitiv: la închidere ele se ating, ricoșează elastic, se despart pentru câteva zeci de microsecunde, se ating din nou, de mai multe ori, înainte de a rămâne în contact ferm. Pentru un om, apăsarea este un eveniment unic care durează o zecime de secundă. Pentru un microcontroler care citește pinul de un milion de ori pe secundă, aceeași apăsare este o rafală de zeci de tranziții, iar un program care numără apăsările va număra, în loc de una, un număr aleatoriu între cinci și cincizeci.

Filtrarea acestor oscilații se poate face hardware, cu un *filtru RC*: o rezistență în serie cu linia și un condensator între linie și masă formează un filtru trece-jos, a cărui constantă de timp $\tau = R \cdot C$ se alege mai mare decât durata oscilațiilor, dar mult mai mică decât durata unei apăsări normale. Cu $R = 10 \text{ k}\Omega$ și $C = 100 \text{ nF}$ se obține $\tau = 1 \text{ ms}$, valoare potrivită pentru un buton obișnuit. Filtrul RC funcționează cu atât mai bine cu cât intrarea are trigger Schmitt, ceea ce la AVR este cazul: fără histerezis, panta lentă produsă de condensator ar genera ea însăși comutări multiple.

Filtrarea se poate face și software, iar în practică aceasta este soluția preferată, pentru că nu costă nimic în componente. Ideea este ca, după detectarea unei tranziții, programul să ignore starea pinului pentru un interval de câteva zeci de milisecunde, timp în care oscilațiile se sting. Implementarea corectă nu folosește o întârziere blocantă, care ar paraliza restul programului, ci compară momentul curent cu momentul ultimei tranziții acceptate.

Exemplu 4.4 Filtrare software a oscilațiilor de contact

```

1  const int BUTON = 2;
2  const unsigned long INTERVAL = 30;    /* milisecunde */
3
4  int stareStabila      = HIGH;
5  int stareAnterioara   = HIGH;
6  unsigned long ultimaSchimbare = 0;
7
8  void setup()
9  {
10     pinMode(BUTON, INPUT_PULLUP);

```

```
11 }
12
13 void loop()
14 {
15     int citire = digitalRead(BUTON);
16
17     if (citire != stareAnterioara) {
18         ultimaSchimbare = millis();    /* a inceput o tranzitie */
19     }
20
21     if ((millis() - ultimaSchimbare) > INTERVAL) {
22         if (citire != stareStabila) {
23             stareStabila = citire;
24             if (stareStabila == LOW) {
25                 /* aici se trateaza apasarea, o singura data */
26             }
27         }
28     }
29
30     stareAnterioara = citire;
31 }
```

Mecanismul este de reținut: fiecare schimbare observată repornește cronometrul, iar starea este declarată stabilă numai după ce a rămas neschimbată un interval întreg. Astfel, rafala de oscilații, oricât de lungă, este tratată ca un singur eveniment.

4.11.2 Relee

Releul este un dispozitiv de comandă folosit pentru a închide și a deschide un alt circuit electric, de obicei unul de putere mult mai mare. El conține o bobină și un set de contacte mecanice; alimentarea bobinei creează un câmp magnetic care atrage o armătură, iar armătura închide sau deschide contactele. Avantajele lui sunt izolarea galvanică între circuitul de comandă și cel comandat, capacitatea de a comuta atât curent continuu cât și alternativ, și rezistența foarte mică a contactului închis.

Un releu nu se comandă niciodată direct de la un pin de microcontroler, din două motive independente. Primul este curentul: bobina unui releu obișnuit consumă între 30 și 100 mA, adică peste maximul absolut al unui pin. Al doilea este tensiunea inversă de autoinducție: în momentul întreruperii curentului prin bobină, energia stocată în câmpul magnetic produce, conform legii inducției, un vârf de tensiune care poate atinge sute de volți, cu polaritate opusă alimentării. Acest vârf ar străpunge instantaneu tranzistorul de comandă și ar ajunge, prin diodele de protecție, la alimentarea microcontrolerului.

Montajul corect folosește deci un tranzistor ca element de comutație, comandat prin bază

sau grilă de la pinul microcontrolerului, și o *diodă de recuperare*, numită și diodă de flyback, montată în paralel cu bobina, cu catodul spre plus. În funcționare normală dioda este blocată și nu face nimic; în momentul întreruperii ea intră în conducție și oferă curentului din bobină un drum închis prin care se poate stinge, limitând vârful de tensiune la o cădere de diodă. Omiterea acestei diode este una dintre cele mai frecvente cauze de distrugere a montajelor care comandă sarcini inductive, iar efectul ei nu apare neapărat imediat, ci după un număr de comutări.

4.11.3 LED-uri și afișoare

Pentru aplicațiile cu multe LED-uri se recomandă folosirea de componente cu un curent de operare de aproximativ 20 mA, valoare care reprezintă un bun compromis între luminozitate și consum. Am văzut deja de ce fiecare LED are nevoie de rezistență proprie de limitare; merită adăugat că legarea mai multor LED-uri în paralel pe aceeași rezistență este o eroare, chiar dacă pare o economie. LED-urile nu au exact aceeași tensiune directă, iar cel cu V_F mai mic va prelua o parte disproporționată din curent, va lumina mai tare, se va încălzi, ceea ce îi va scădea și mai mult V_F , într-un proces care duce la distrugerea lui.

Când numărul de LED-uri depășește numărul de pini disponibili, soluția clasică este *multiplexarea*. La un afișor cu mai multe cifre cu șapte segmente, segmentele omoloage ale tuturor cifrelor sunt legate împreună, iar catodii comuni ai cifrelor sunt comandați individual. Programul afișează pe segmente configurația primei cifre, activează catodul primei cifre, așteaptă un timp scurt, o stinge, trece la a doua și tot așa, ciclând suficient de repede încât ochiul, din cauza persistenței retiniene, să vadă toate cifrele aprinse simultan. O frecvență de reîmprospătare de peste aproximativ 50 Hz pe cifră este suficientă pentru a elimina senzația de pâlpâire.

Multiplexarea are un preț: fiecare cifră este aprinsă doar o fracțiune din timp, deci luminozitatea medie scade proporțional cu numărul de cifre. Se compensează crescând curentul instantaneu prin segmente peste valoarea continuă admisă, ceea ce LED-urile suportă atât timp cât curentul mediu rămâne în limite, dar asta cere din nou drivere externe, pentru că pinii microcontrolerului nu pot da acel curent.

4.11.4 Optocuploare

Optocuplorul este componenta folosită atunci când se cere *izolare galvanică*, adică absența oricărei legături conductive între circuitul de comandă și cel comandat. El conține, în aceeași capsulă, un LED și un element fotosensibil, de obicei un fototranzistor, separate printr-o barieră izolatoare transparentă. Curentul prin LED produce lumină, lumina deschide fototranzistorul, iar informația trece astfel dintr-un circuit în altul fără ca electronii să treacă vreodată.

Utilitatea izolării este dublă. Pe de o parte, ea protejează microcontrolerul și, prin el, calculatorul conectat la portul USB, de tensiuni periculoase din partea de putere: un montaj care comută 230 V poate avea, în caz de defect, întreaga tensiune de rețea pe partea comandată, iar optocuplorul o oprește. Pe de altă parte, izolarea rupe buclele de masă, adică

acele trasee prin care curenții mari de retur ai unui motor sau ai unei sarcini comutate ar ajunge să circule prin masa circuitului logic, producând căderi de tensiune parazite și citiri eronate.

Din punctul de vedere al portului, un optocuplor este pur și simplu un LED și se tratează ca atare, cu rezistență de limitare calculată după aceeași formulă. Pe partea de ieșire, fototranzistorul se comportă ca un comutator care leagă linia la masă, deci se folosește în mod natural împreună cu o rezistență de pull-up, la fel ca un buton. Este motivul pentru care slide-urile cursului menționează optocuplorul alături de butoane și comutatoare ca aplicație tipică a rezistenței de pull-up.

4.11.5 Ce reține proiectantul

Trecând peste diversitatea circuitelor prezentate, se observă că toate răspund aceluiași câteva întrebări. Cine impune nivelul liniei când nu îl impune nimeni altcineva, și deci unde este nevoie de pull-up. De unde vine curentul necesar sarcinii, și deci unde este nevoie de un element intermediar de putere. Ce se întâmplă cu energia stocată într-o sarcină inductivă la deconectare, și deci unde este nevoie de o diodă de recuperare. Cât timp durează evenimentul pe care vrem să îl detectăm față de perioada de eșantionare, și deci unde este nevoie de întrerupere în loc de interogare. Un pin de microcontroler este ieftin, dar nu este gratuit și, mai ales, nu este nici sursă de putere, nici izolator, nici filtru. Tot ce depășește rolul lui de element logic trebuie construit în jurul lui.

Întrebări de verificare

1. Explicați de ce un port AVR are nevoie de trei registre distincte și descrieți situația concretă în care valoarea din `PORTx` diferă de cea citită din `PINx` pentru același pin.
2. Un pin este configurat cu `DDRD` bitul 4 pe 0 și `PORTD` bitul 4 pe 1. În ce regim se află pinul, ce nivel se citește în repaus și ce se întâmplă dacă un contact exterior îl leagă la masă?
3. Descrieți structura etajului de ieșire în contratimp și explicați de ce cele două tranzistoare nu au voie să conducă simultan.
4. Explicați de ce citirea unui pin este întârziată față de schimbarea fizică a tensiunii pe el și estimați această întârziere pentru un ATmega328P la 16 MHz.
5. Ce este metastabilitatea, prin ce mecanism intern al bistabilului se explică și de ce nu poate fi eliminată complet, ci doar făcută improbabilă?
6. Comparați rolul triggerului Schmitt cu rolul sincronizatorului cu două etaje. Care dintre cele două cauze ale metastabilității rezolvă fiecare?
7. Un pin de intrare este lăsat nelegat. Descrieți ce tensiune apare pe el, de ce citirile par aleatorii și ce efect are situația asupra consumului microcontrolerului.

8. Enumerați trei situații concrete în care pull-up-ul intern al AVR-ului nu este suficient și explicați pentru fiecare de ce este nevoie de o rezistență externă.
9. Un LED verde cu $V_F = 2,1$ V trebuie alimentat la 12 mA de la un pin la 5 V. Calculați rezistența de limitare, alegeți valoarea standard și determinați puterea disipată pe rezistență.
10. Explicați diferența de cost în cicli de ceas între `PORTB |= (1 << PB5)` și `digitalWrite(13, HIGH)` și precizați o situație în care diferența contează efectiv.
11. De ce activarea unui periferic precum UART sau SPI face ca pinii corespunzători să nu mai poată fi folosiți ca pini cu scop general? Dați un exemplu de conflict pe placa Arduino Uno.
12. Explicați de ce un releu nu poate fi comandat direct de la un pin și care este rolul diodei montate în paralel cu bobina.

5. Sistemul de întreruperi

Prezentare generală
Caracteristici ale sistemului de întreruperi
Principiul de funcționare
Rutina de tratare a întreruperii
Tabelul vectorilor de întrerupere
Surse de întrerupere
Priorități și imbricarea întreruperilor
Variabile partajate și condiții de cursă
Aplicație cu sistemul de întreruperi
Întrebări de verificare

5.1 Prezentare generală

Până acum programele scrise pentru microcontroler au avut o structură simplă și previzibilă: o secvență de inițializare, urmată de o buclă infinită în care procesorul citește intrările, calculează ceva și comandă ieșirile. Ordinea în care se întâmplă lucrurile este dictată în întregime de programator, iar procesorul nu face niciodată altceva decât ceea ce urmează în listă. Această structură funcționează foarte bine atâta timp cât lumea din jurul microcontrolerului are răbdare să fie întrebăată. Problema apare în momentul în care lumea exterioară nu mai are răbdare: un buton este apăsat pentru câteva zecimi de milisecundă, un octet sosește pe linia serială și trebuie citit înainte să vină următorul, un senzor de turație generează un impuls scurt la fiecare trecere a unei palete prin dreptul lui. În toate aceste situații evenimentul nu așteaptă ca programul să ajungă la instrucțiunea care îl verifică.

Mecanismul prin care microcontrolerul răspunde unor astfel de evenimente se numește *întrerupere* (în engleză *interrupt*). Prin întrerupere înțelegem suspendarea temporară a procesului normal de execuție a programului, pentru rezolvarea unei probleme considerate prioritare, urmată de reluarea execuției exact din punctul în care aceasta a fost suspendată. Cuvântul cheie din definiție este „suspendarea”: programul principal nu este abandonat și nu este repornit, ci doar pus în pauză pentru o durată foarte scurtă, fără ca el să afle vreodată că acest lucru s-a întâmplat.

5.1.1 Alternativa: interogarea ciclică

Înainte de a înțelege de ce avem nevoie de întreruperi, merită să vedem cum se descurcă un program fără ele. Metoda clasică se numește *interogare ciclică* sau, cu termenul englezesc folosit peste tot în documentație, *polling*. Ideea este simplă: în bucla principală programul verifică periodic starea fiecărei surse de evenimente care îl interesează și, dacă găsește ceva schimbat, reacționează.

Exemplu 5.1 Structura tipică a unui program cu interogare ciclică

```

1  int main(void)
2  {
3      DDRB |= (1 << PB5);           // LED-ul de pe PB5 ca ieseire
4      DDRD &= ~(1 << PD2);         // butonul pe PD2 ca intrare
5      PORTD |= (1 << PD2);         // rezistentă internă de pull-up
6
7      uint8_t stare_veche = 1;
8
9      while (1)
10     {
11         uint8_t stare_noua = (PIND & (1 << PD2)) ? 1 : 0;
12
13         if (stare_veche == 1 && stare_noua == 0)    // front
14             ↪ cazator
15             {
16                 PORTB ^= (1 << PB5);                // comută LED-ul
17                 stare_veche = stare_noua;
18
19                 citește_senzor_temperatura();        // durează câteva
20                 ↪ milisecunde
21                 actualizează_afisajul();             // durează și mai mult
22     }

```

Programul din listingul anterior funcționează, dar are două defecte pe care nu le poate corecta nicio rescriere ingenioasă a buclei. Primul defect este risipa de timp. Procesorul citește registrul PIND de zeci de mii de ori pe secundă, iar în marea majoritate a cazurilor răspunsul este „nu s-a întâmplat nimic”. Fiecare astfel de verificare consumă cicluri de ceas, adică energie și, mai ales, timp care ar fi putut fi folosit pentru altceva. Într-o aplicație alimentată de la baterie situația este și mai neplăcută, pentru că un procesor care întreabă încontinuu nu poate fi trecut niciodată într-un regim de consum redus.

Al doilea defect este mult mai grav, pentru că duce la pierderea informației. Între două verificări succesive ale butonului, programul execută citirea senzorului și actualizarea afișajului. Dacă aceste două operații durează, să zicem, cinci milisecunde, atunci starea butonului este privită o dată la cinci milisecunde. Un impuls mai scurt de cinci milisecunde poate cădea în întregime în intervalul dintre două priviri și programul nu va afla niciodată de existența lui. Evenimentul a existat fizic, dar nu a fost observat.

Exemplu

Un senzor optic montat pe axul unui motor generează un impuls de nivel jos cu durata de $40\ \mu\text{s}$ la fiecare rotație. Bucla principală a programului, compilată pentru un ATmega328P la 16 MHz, are nevoie de 1200 de cicluri de ceas pentru un parcurs complet. Poate fi detectat impulsul prin interogare ciclică?

Rezolvare. La 16 MHz, un ciclu de ceas durează $T = 1/16 \cdot 10^6 = 62,5\ \text{ns}$. Un parcurs complet al buclei durează deci $1200 \times 62,5\ \text{ns} = 75\ \mu\text{s}$. Impulsul de $40\ \mu\text{s}$ este mai scurt decât intervalul dintre două citiri consecutive ale pinului, așa că el poate începe și se poate termina complet între două priviri ale programului. Detectia nu este imposibilă, dar este aleatoare: uneori impulsul va fi surprins, alteori nu, iar turația măsurată va fi greșită într-un mod imprevizibil. Pentru o detecție sigură prin interogare ar trebui ca bucla să se închidă în mai puțin de $40\ \mu\text{s}$, adică în mai puțin de 640 de cicluri, ceea ce ar lăsa foarte puțin loc pentru restul aplicației.

5.1.2 Analogia cu soneria

Diferența dintre cele două abordări se înțelege cel mai bine prin analogia cu ușa unei case. Presupunem că așteptăm un colet și că ușa nu are sonerie. Singura soluție este să mergem din când în când până la ușă și să verificăm dacă a venit cineva. Aceasta este interogarea ciclică. Dacă mergem des, pierdem tot timpul pe hol și nu mai apucăm să facem nimic altceva. Dacă mergem rar, curierul apucă să sune, să aștepte puțin și să plece, iar noi găsim ușa goală. Nu există o frecvență a plimbărilor care să rezolve ambele probleme simultan.

Soluția reală este soneria. Cu o sonerie montată, ne putem vedea liniștiți de treabă în cealaltă cameră, pentru că nu mai avem nicio obligație să verificăm ușa. În momentul în care cineva apasă butonul soneriei, suntem anunțați, lăsăm din mână ceea ce făceam, mergem la ușă, preluăm coletul și ne întoarcem exact la activitatea întreruptă. Soneria este exact ceea ce face sistemul de întreruperi al microcontrolerului: procesorul nu monitorizează apariția unei întreruperi, ci reacționează doar atunci când întreruperea apare. Este important de reținut că evenimentul nu este generat de procesor, ci de un modul periferic sau de mediul exterior; procesorul este partea care primește anunțul.

Analogia are și o continuare utilă pentru mai târziu în acest capitol. Dacă, în timp ce mergem spre ușă, decidem să ne oprim ca să terminăm de spălat vasele, coletul va aștepta mult și curierul va pleca. La fel, o rutină de tratare a întreruperii care face prea multe lucruri blochează întregul sistem. Iar dacă, în timp ce vorbim cu curierul la ușă, sună telefonul, soneria și telefonul se suprapun; ce facem în acest caz este exact problema priorităților.

5.2 Caracteristici ale sistemului de întreruperi

Un sistem de întreruperi, indiferent de familia de microcontrolere în care este implementat, are un set de proprietăți comune care merită enumerate înainte de a intra în detaliile arhitecturii AVR. Ele explică de ce mecanismul este atât de răspândit și, în același timp, de ce trebuie folosit cu grijă.

Întreruperea este, de regulă, generată ca răspuns la un efect fizic, intern sau extern, al unui modul periferic. Exemple tipice sunt schimbarea nivelului logic al unui pin de intrare, încheierea unei perioade de timp măsurate de un numărător, terminarea unei operații de transmisie sau de recepție de date pe o interfață serială, sau sfârșitul unei conversii analog-numerică. Ceea ce au în comun toate aceste evenimente este caracterul lor *asincron*: ele se produc la momente de timp care nu sunt corelate cu poziția curentă a programului. Tocmai de aceea creșterea de eficiență adusă de întreruperi este atât de mare: programul nu mai trebuie construit în jurul momentelor în care evenimentele s-ar putea produce.

Tratarea unei întreruperi presupune existența unei subrutine dedicate, care conține codul ce trebuie executat atunci când evenimentul apare. Această subrutină, numită rutină de tratare, nu este apelată niciodată explicit din program; ea este invocată automat de hardware. Din punctul de vedere al programatorului, funcția respectivă este scrisă o singură dată și apoi „uitată”, în sensul că nimic din codul vizibil nu conține un apel către ea.

Apariția unui eveniment de întrerupere conduce, la nivel hardware, la setarea unui bistabil numit *indicator de întrerupere* sau, cu termenul din documentație, *flag*. Acest bistabil memorează faptul că evenimentul s-a produs, chiar dacă evenimentul fizic a durat foarte puțin. Existența indicatorului este exact ceea ce rezolvă problema impulsurilor scurte discutate mai devreme: impulsul de $40\ \mu\text{s}$ poate să dispară, dar bitul care îl marchează rămâne setat până când este tratat sau șters explicit.

Sistemul de întreruperi poate fi activat și dezactivat, atât global, cât și individual, pe fiecare sursă în parte. Un procesor AVR are un bit de activare globală, notat I, în registrul de stare SREG, iar fiecare periferic are propriul său bit de activare locală. O cerere de întrerupere este acceptată numai dacă toate condițiile sunt îndeplinite simultan: indicatorul sursei este setat, activarea locală a sursei este pornită și activarea globală este pornită. Figura 5.1 arată această condiție sub forma unei porți logice.

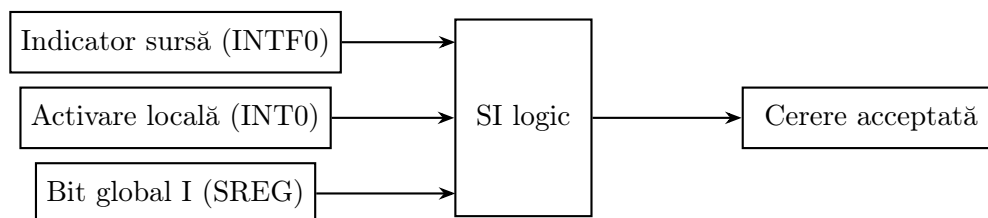


Figura 5.1 Cele trei condiții care trebuie îndeplinite simultan pentru ca o cerere de întrerupere să fie acceptată de procesor.

Întreruperile pot avea priorități diferite, iar ordinea în care sunt tratate atunci când apar simultan este stabilită fie de hardware, fie de programator, în funcție de arhitectură.

Vom reveni asupra acestui aspect într-o secțiune separată, pentru că pe AVR el are o formă particulară, adesea neînțeleasă corect.

În sfârșit, o proprietate care ține mai degrabă de proiectarea aplicației decât de hardware: sistemul de întreruperi permite construirea unor programe care reacționează în timp real la stimuli externi. Prin „timp real” nu înțelegem „foarte repede”, ci „într-un interval de timp garantat”. Un program bazat pe întreruperi poate oferi o astfel de garanție, pentru că timpul scurs între apariția evenimentului și începutul tratării lui poate fi calculat, în timp ce un program bazat pe interogare oferă doar o medie, cu abateri imprevizibile.

5.3 Principiul de funcționare

Vom urmări acum, pas cu pas, ce se întâmplă mecanic în interiorul procesorului atunci când apare o întrerupere. Descrierea este dată pentru arhitectura AVR, dar succesiunea este aproape identică pe orice microcontroler de opt sau de treizeci și doi de biți.

Punctul de plecare este programul principal, notat în literatura de specialitate MPF (*Main Program Flow*). Procesorul execută instrucțiune după instrucțiune, iar registrul numit *contor de program* (PC, de la *Program Counter*) conține în permanență adresa instrucțiunii care urmează să fie executată. Presupunem că, la un moment oarecare, un periferic setează indicatorul său de întrerupere și că cele trei condiții din Figura 5.1 sunt îndeplinite.

5.3.1 Succesiunea de operații

Primul lucru pe care îl face procesorul este să **termine instrucțiunea aflată în curs de execuție**. Acesta nu este un detaliu neînsemnat, ci o garanție fundamentală de corectitudine. O instrucțiune AVR poate dura de la un ciclu de ceas până la patru sau cinci cicluri, iar oprirea ei la jumătate ar lăsa procesorul într-o stare inconsistentă: o scriere în memorie efectuată doar parțial, sau un rezultat aritmetic calculat, dar nedepus în registrul destinație. De aceea întreruperea nu este niciodată luată în considerare la mijlocul unei instrucțiuni, ci doar la granița dintre două instrucțiuni. Consecința practică este că o instrucțiune lungă adaugă întârziere la răspunsul la întrerupere.

În al doilea rând, procesorul **salvează contorul de program pe stivă**, operație notată prescurtat **push(PC)**. Ceea ce se salvează este adresa instrucțiunii următoare din programul principal, adică exact locul din care execuția va trebui reluată. Pe ATmega328P adresa are nevoie de doi octeți, deci indicatorul de stivă SP scade cu două poziții. Faptul că adresa de revenire este pusă pe stivă, și nu într-un registru dedicat, are o consecință importantă: dacă stiva este coruptă sau dacă se suprapune peste zona de variabile, revenirea din întrerupere va duce programul într-un loc arbitrar din memoria de program, iar comportamentul devine complet imprevizibil.

În al treilea rând, procesorul **dezactivează automat întreruperile**, ștergând bitul I din registrul SREG. Este echivalentul hardware al gestului de a scoate din priză soneria în timp ce discutăm cu curierul. Fără acest pas, o a doua întrerupere ar putea sosi imediat după prima, ar salva încă o adresă pe stivă, ar fi la rândul ei întreruptă și așa mai departe; într-o situație nefericită, stiva ar crește necontrolat până când ar acoperi variabilele programului.

Ștergerea automată a bitului I garantează că rutina de tratare începe într-un mediu liniștit.

În al patrulea rând, procesorul **încarcă în contorul de program adresa vectorului** corespunzător sursei care a generat cererea. Vectorul este preluat din tabelul vectorilor de întrerupere, despre care vom vorbi în secțiunea următoare, iar alegerea intrării din tabel se face pe baza sursei. În acest moment execuția a fost deviată: următoarea instrucțiune executată nu mai aparține programului principal, ci rutinei de tratare.

În al cincilea rând, se **execută rutina de tratare a întreruperii**. Pe durata execuției ei, programul principal este pur și simplu oprit; nu rulează în paralel, nu rulează mai încet, ci nu rulează deloc. Toate operațiile din rutină, inclusiv cele inutile, se adaugă direct la timpul în care restul aplicației nu face nimic.

În sfârșit, rutina se încheie cu instrucțiunea **reti** (*return from interrupt*). Aceasta face două lucruri distincte: extrage de pe stivă adresa salvată și o pune înapoi în contorul de program, apoi setează la loc bitul I din **SREG**, redeschizând sistemul de întreruperi. Diferența față de instrucțiunea obișnuită de revenire dintr-o subrutină, **ret**, este exact această reacție la întreruperilor. Din acest motiv o rutină de tratare nu se poate termina niciodată cu **ret**: programul ar reveni corect, dar ar rula mai departe cu întreruperile stinse, iar sistemul ar părea că „se blochează” după prima întrerupere.

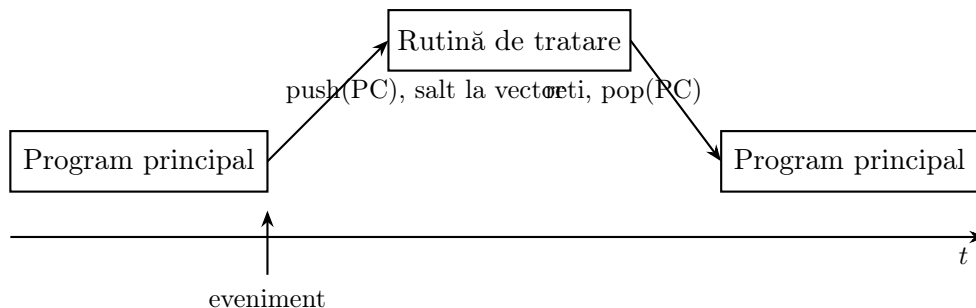


Figura 5.2 Cronograma execuției: programul principal este suspendat la apariția evenimentului, rutina de tratare se execută, iar apoi execuția revine exact în punctul părăsit.

Figura 5.2 rezumă întreaga succesiune. Este util de observat că, privit din interiorul programului principal, nimic nu s-a schimbat: aceleași registre conțin aceleași valori, aceiași indicatori de condiție sunt setați, iar execuția continuă cu instrucțiunea care urma oricum. Această iluzie de continuitate este esența întregului mecanism, iar menținerea ei costă efort, așa cum vom vedea imediat.

5.3.2 Latența răspunsului la întrerupere

Timpul scurs între momentul în care evenimentul se produce și momentul în care prima instrucțiune utilă din rutina de tratare începe să se execute se numește *latență*. Ea nu este nulă și nici constantă, iar componentele ei sunt următoarele: timpul de terminare a instrucțiunii curente, numărul fix de cicluri necesare procesorului pentru a recunoaște cererea și a salva contorul de program, timpul necesar executării instrucțiunii de salt din tabelul vectorilor, și în sfârșit timpul consumat de prologul rutinei, adică de salvarea contextului.

Exemplu

Un ATmega328P funcționează la 16 MHz. Care este ordinul de mărime al latenței minime de răspuns la o întrerupere externă, dacă procesorul recunoaște cererea în patru perioade de ceas, iar saltul din tabelul vectorilor durează trei perioade?

Rezolvare. O perioadă de ceas durează $T = 1/(16 \cdot 10^6) = 62,5$ ns. Cele patru perioade de recunoaștere și salvare a contorului de program dau $4 \times 62,5 = 250$ ns, iar saltul adaugă $3 \times 62,5 = 187,5$ ns, deci în total aproximativ 437,5 ns. La acest interval se adaugă terminarea instrucțiunii aflate în curs, care poate însemna încă una până la patru perioade, și prologul generat de compilator, care pe o rutină cu mai multe variabile poate depăși douăzeci de perioade. Concluzia practică este că latența reală este de ordinul unei microsecunde, nu al zecilor de nanosecunde, și că ea variază de la o apariție la alta. Pentru măsurători de timp precise nu ne bazăm niciodată pe momentul intrării în rutină, ci pe valoarea capturată de hardware, de exemplu de unitatea de captură a numărătorului.

5.4 Rutina de tratare a întreruperii

Rutina de tratare a întreruperii, cunoscută în literatura de specialitate sub numele de ISR (*Interrupt Service Routine*), este o funcție specială din cadrul programului, care este invocată automat atunci când apare întreruperea corespunzătoare. Ea se scrie ca orice altă funcție, dar se comportă complet diferit, iar diferențele merită discutate pe rând.

5.4.1 Salvarea și restaurarea contextului

Am spus mai devreme că, la revenirea din întrerupere, programul principal trebuie să găsească procesorul exact în starea în care l-a lăsat. Hardware-ul singur nu poate garanta acest lucru. El salvează contorul de program, pentru că fără el nu ar ști unde să revină, dar nu salvează nimic altceva. Toate celelalte resurse ale procesorului rămân la dispoziția rutinei de tratare, iar dacă rutina le modifică, programul principal se va trezi cu ele schimbate.

Cea mai delicată dintre aceste resurse este registrul de stare **SREG**, care conține indicatorii de condiție: zero, transport, semn, depășire. Să presupunem că programul principal tocmai a executat o comparație și urmează să execute un salt condiționat pe baza rezultatului acesteia. Dacă între cele două instrucțiuni apare o întrerupere, iar rutina de tratare execută la rândul ei orice operație aritmetică, indicatorii vor fi rescriși. La revenire, saltul condiționat va fi luat pe baza rezultatului calculului din rutina de tratare, nu pe baza comparației din programul principal. Rezultatul este o eroare imposibil de reprodus prin depanare pas cu pas, pentru că depinde de momentul exact în care a sosit întreruperea.

Aceeași problemă apare cu registrele de lucru. Compilatorul ține valorile intermediare ale programului principal în registrele generale ale procesorului; dacă rutina de tratare folosește aceleași registre pentru propriile calcule, valorile programului principal sunt distruse.

Ansamblul format din registrul de stare și din registrele de lucru folosite se numește *context*. Salvarea contextului la intrarea în rutină și restaurarea lui la ieșire sunt obligatorii. În limbaj de asamblare, programatorul le scrie de mână: o secvență de instrucțiuni **push** la

început și o secvență simetrică de **pop** la sfârșit, în ordine inversă. Este o muncă mecanică și foarte ușor de greșit, iar o singură omisiune produce o eroare care apare o dată la câteva ore de funcționare.

Din fericire, în C acest lucru nu mai trebuie făcut manual. Macroul **ISR()** din biblioteca **avr/interrupt.h** îi spune compilatorului că funcția respectivă este o rutină de tratare, iar compilatorul generează automat prologul și epilogul: salvează **SREG**, salvează exact acele registre pe care corpul funcției le folosește efectiv, execută corpul, restaurează totul în ordine inversă și încheie cu **reti** în loc de **ret**. Faptul că salvează doar registrele folosite este important, pentru că o rutină simplă produce un prolog scurt, în timp ce o rutină care apelează alte funcții obligă compilatorul să salveze mult mai mult, neștiind ce anume folosesc acele funcții.

Exemplu 5.2 Forma unei rutine de tratare în C, pentru întreruperea externă INT0

```

1 #include <avr/io.h>
2 #include <avr/interrupt.h>
3
4 ISR(INT0_vect)
5 {
6     PORTB ^= (1 << PB5);    // corpul rutinei: o singura operatie
7 }
```

Numele scris între paranteze, **INT0_vect**, nu este un parametru oarecare, ci numele simbolic al vectorului de întrerupere. El este definit în fișierele antet ale bibliotecii pentru microcontrolerul ales și leagă funcția de poziția corespunzătoare din tabelul vectorilor. O greșeală de scriere a acestui nume nu produce, în general, o eroare de compilare vizibilă; produce în schimb o rutină care nu este niciodată apelată, iar la apariția întreruperii programul sare într-o intrare netratată din tabel și se resetează.

5.4.2 Reguli de scriere a unei rutine de tratare

Pe durata execuției rutinei de tratare, programul principal este oprit și, pe AVR, celelalte întreruperi sunt și ele blocate. Orice ciclu de ceas consumat inutil în rutină este un ciclu în care aplicația nu răspunde la nimic altceva. De aici decurg câteva reguli practice, pe care este bine să le tratăm ca pe niște obligații și nu ca pe niște sugestii.

Rutina trebuie să fie scurtă. Idealul este de câteva zeci de instrucțiuni: setarea unui indicator, citirea unui registru de date, incrementarea unui numărător. Prelucrarea propriu-zisă se face în bucla principală, care citește indicatorul și reacționează atunci când are timp. Această împărțire a muncii, în care rutina doar constată evenimentul iar programul principal îl prelucrează, este tiparul de proiectare cel mai important din tot capitoul.

În rutină nu se folosesc funcții de întârziere. Un apel **_delay_ms(100)** sau **delay(100)** într-o rutină de tratare oprește întregul sistem pentru o zecime de secundă. În acest interval nu mai este recepționat niciun caracter pe linia serială, nu mai este numărat niciun impuls de la senzor și nu mai este actualizat niciun numărător de timp.

În rutină nu se fac operații lente. Trimiterea unui șir de caractere pe linia serială, scrierea în EEPROM, calculele în virgulă mobilă și afișarea pe un ecran cu cristale lichide sunt toate operații care durează milisecunde. Ele nu au ce căuta într-o rutină de tratare. Tentația de a scrie un mesaj de depanare cu `Serial.print()` chiar în interiorul rutinei este mare și este exact greșeala care transformă un sistem funcțional într-unul care pierde date.

Rutina nu returnează valori și nu primește parametri. Nu există cine să îi transmită argumente și nu există cine să îi preia rezultatul, pentru că apelul nu vine din program, ci din hardware. Comunicarea cu restul aplicației se face exclusiv prin variabile globale, ceea ce ne duce direct la problema discutată în secțiunea despre condiții de cursă.

Observație

Pe platforma Arduino, funcțiile `millis()` și `micros()` se bazează pe o întrerupere de numărător. Deoarece întreruperile sunt dezactivate în interiorul unei rutine de tratare, valoarea returnată de `millis()` nu avansează pe durata execuției acesteia. Funcția poate fi citită, și este utilă de exemplu pentru marcarea momentului evenimentului, dar nu poate fi folosită pentru a măsura o durată în interiorul rutinei și cu atât mai puțin pentru a aștepta. Funcția `delay()` plasată într-o rutină de tratare nu se termină niciodată.

5.5 Tabelul vectorilor de întrerupere

Am spus că, la acceptarea unei cereri, procesorul încarcă în contorul de program adresa vectorului corespunzător sursei. Legătura dintre sursa fizică a întreruperii și rutina care o tratează este realizată de o structură de date aflată în memorie, numită *tabelul vectorilor de întrerupere*.

5.5.1 Ce este și unde se află

Tabelul vectorilor este o secțiune fixă de memorie, rezervată la o adresă impusă de arhitectura microcontrolerului. Pe AVR el se află la începutul memoriei de program, adică în Flash, începând de la adresa zero. Fiecare intrare din tabel corespunde unui anumit tip de întrerupere, iar poziția intrării în tabel este stabilită de producător și nu poate fi schimbată de programator.

Pe microcontrolerele cu memorie de program mică, fiecare intrare conține o singură instrucțiune de salt relativ, `rjmp`, care ocupă un cuvânt de șaisprezece biți. Pe ATmega328P, care are 32 kocteți de Flash, distanța până la rutina de tratare poate depăși raza de acțiune a unui salt relativ, așa că fiecare intrare conține o instrucțiune `jmp` completă, care ocupă două cuvinte. De aceea vectorii se află din două în două cuvinte: RESET la adresa 0x0000, INT0 la 0x0002, INT1 la 0x0004 și așa mai departe.

Structura din Figura 5.3 explică de ce tabelul conține salturi și nu direct codul rutinelor. Spațiul dintre doi vectori consecutivi este de numai două cuvinte, insuficient pentru orice rutină reală. Prin urmare intrarea din tabel nu este destinația finală, ci un indicator către ea: procesorul sare în tabel, iar de acolo saltul îl duce oriunde în Flash, acolo unde compilatorul a plasat corpul funcției.

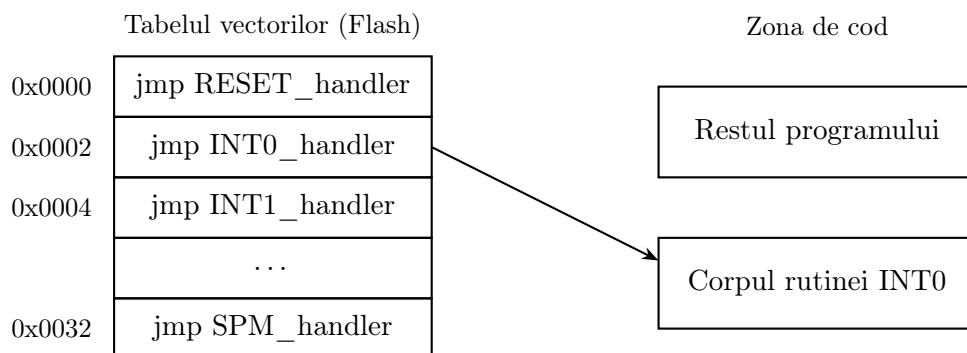


Figura 5.3 Structura tabelului vectorilor de întrerupere pe ATmega328P: fiecare intrare conține un salt către rutina de tratare, plasată oriunde în memoria de program.

Prima intrare, cea de la adresa zero, este vectorul de RESET. El nu corespunde unei întreruperi propriu-zise, ci punctului de pornire al microcontrolerului după alimentare, după o resetare externă sau după expirarea numărătorului de gardă. Faptul că se află în același tabel are o consecință utilă: dacă o întrerupere apare fără să existe o rutină scrisă pentru ea, intrarea implicită generată de compilator trimite execuția la vectorul de RESET, iar programul repornește de la început. Un microcontroler care se resetează la intervale aparent aleatoare este, de foarte multe ori, un microcontroler care are o sursă de întrerupere activată și netratată.

La unele familii de microcontrolere, în special la cele pe treizeci și doi de biți, adresele memorate în tabelul vectorilor sunt configurabile, iar tabelul însuși poate fi mutat în memoria de date. Acest lucru permite schimbarea rutinei de tratare în timpul funcționării, de exemplu pentru a folosi rutine diferite în etape diferite ale aplicației. Pe AVR tabelul este fix, cu excepția posibilității de a-l muta la începutul secțiunii de încărcare, opțiune folosită de programele de tip bootloader.

5.5.2 Vectorii de întrerupere ai microcontrolerului ATmega328P

Tabelul 5.1 prezintă principalele intrări din tabelul vectorilor pentru ATmega328P, în ordinea în care apar în memorie. Adresele sunt exprimate în cuvinte, așa cum sunt date în fișa de catalog.

Se observă imediat o regularitate: sursele care necesită reacție rapidă și care sunt legate direct de lumea exterioară, adică întreruperile externe, se află în capul tabelului, în timp ce sursele lente, legate de memorie, se află la sfârșit. Această ordonare nu este întâmplătoare, pentru că pe AVR poziția în tabel are un rol suplimentar, pe care îl discutăm în secțiunea despre priorități.

5.6 Surse de întrerupere

Sursele de întrerupere ale unui microcontroler se împart în două categorii, după originea evenimentului: surse externe, în care evenimentul vine de pe un pin, și surse interne, în care evenimentul este produs de un modul din interiorul cipului.

Tabela 5.1 Principalele surse de întrerupere ale microcontrolerului ATmega328P și vectorii corespunzători.

Nr.	Adresă	Sursă	Semnificație
1	0x0000	RESET	Alimentare, resetare externă, numărător de gardă
2	0x0002	INT0	Cerere de întrerupere externă 0 (pinul PD2)
3	0x0004	INT1	Cerere de întrerupere externă 1 (pinul PD3)
4	0x0006	PCINT0	Schimbare de nivel pe portul B
5	0x0008	PCINT1	Schimbare de nivel pe portul C
6	0x000A	PCINT2	Schimbare de nivel pe portul D
7	0x000C	WDT	Expirarea numărătorului de gardă
8	0x000E	TIMER2_COMPA	Comparare A la numărătorul 2
9	0x0010	TIMER2_COMPB	Comparare B la numărătorul 2
10	0x0012	TIMER2_OVF	Depășire la numărătorul 2
11	0x0014	TIMER1_CAPT	Captură de eveniment la numărătorul 1
12	0x0016	TIMER1_COMPA	Comparare A la numărătorul 1
13	0x0018	TIMER1_COMPB	Comparare B la numărătorul 1
14	0x001A	TIMER1_OVF	Depășire la numărătorul 1
15	0x001C	TIMER0_COMPA	Comparare A la numărătorul 0
16	0x001E	TIMER0_COMPB	Comparare B la numărătorul 0
17	0x0020	TIMER0_OVF	Depășire la numărătorul 0
18	0x0022	SPI_STC	Transfer serial SPI încheiat
19	0x0024	USART_RX	Recepție încheiată pe USART
20	0x0026	USART_UDRE	Registrul de date USART este liber
21	0x0028	USART_TX	Transmisie încheiată pe USART
22	0x002A	ADC	Conversie analog-numerică încheiată
23	0x002C	EE_READY	Memoria EEPROM este pregătită
24	0x002E	ANALOG_COMP	Comparator analogic
25	0x0030	TWI	Interfața serială cu două fire (I2C)
26	0x0032	SPM_READY	Scriere în memoria de program încheiată

5.6.1 Întreruperi externe

Întreruperile externe sunt generate de intrările digitale ale microcontrolerului, adică de pini cu funcții generale care au și capacitate de întrerupere. Un pin obișnuit de intrare poate fi citit, dar nu poate anunța nimic; un pin cu capacitate de întrerupere are, în plus, un circuit de detecție care compară nivelul curent cu cel anterior sau cu un nivel de referință și care setează indicatorul corespunzător.

Detecția se poate face în două moduri fundamental diferite. În modul *activ pe nivel* (*level-triggered*), cererea de întrerupere există în permanență atâta timp cât linia se află în starea considerată activă, de obicei nivelul jos. Acest mod are o consecință care surprinde adesea: dacă rutina de tratare se termină iar linia este încă jos, cererea reapare imediat și rutina este executată din nou, la nesfârșit, până când sursa exterioară eliberează linia. Modul activ pe nivel este util atunci când mai multe periferice partajează aceeași linie de cerere printr-o conexiune de tip colector în gol, pentru că permite tratarea lor succesivă, dar este periculos în utilizarea naivă.

În modul *activ pe front* (*edge-triggered*), cererea apare o singură dată, la momentul tranziției, și este memorată în indicator. Se poate alege frontul crescător, adică trecerea de la nivelul jos la cel înalt, frontul căzător, adică trecerea inversă, sau orice schimbare de nivel. Acesta este modul folosit în majoritatea aplicațiilor, pentru că evenimentul este prin natura lui punctual: butonul a fost apăsător, paleta a trecut prin dreptul senzorului, semnalul de sincronizare a sosit.

Pe ATmega328P există două întreruperi externe complete, INT0 pe pinul PD2 și INT1 pe pinul PD3, al căror mod de declanșare se configurează în registrul EICRA prin perechile de biți ISC01/ISC00 și, respectiv, ISC11/ISC10. Combinațiile sunt următoarele: 00 pentru nivel jos, 01 pentru orice schimbare de nivel, 10 pentru front căzător și 11 pentru front crescător. Activarea individuală se face prin biții INT0 și INT1 din registrul EIMSK, iar indicatorii se găsesc în registrul EIFR, sub numele INTF0 și INTF1.

Pe lângă acestea, microcontrolerul dispune de întreruperi de tip *pin change*, notate PCINT. Ele acoperă practic toți pinii, dar sunt mult mai grosiere: nu se poate alege frontul, detecția se face pe orice schimbare, iar toți pinii unui port partajează un singur vector. Rutina de tratare trebuie deci să citească portul și să compare cu valoarea anterioară pentru a afla care pin s-a modificat și în ce sens. În schimb, ele rezolvă situația în care avem nevoie de multe intrări cu capacitate de întrerupere și doar două întreruperi externe complete.

O a doua categorie importantă de surse externe o formează perifericele de comunicație: USART, SPI, interfața cu două fire I2C și, pe microcontrolere mai complexe, USB sau Ethernet. Acestea generează întreruperi atunci când datele sunt gata de a fi transferate sau atunci când au fost primite, permițând procesorului să le prelucreze imediat. Utilitatea lor este evidentă în cazul recepției seriale: la o viteză de 115200 de biți pe secundă, un caracter de zece biți sosește la fiecare 87 de microsecunde aproximativ, iar registrul de recepție trebuie golit înainte de sosirea următorului. Fără întrerupere, programul principal ar trebui să verifice registrul de stare cu o frecvență cel puțin dublă, ceea ce ar face imposibilă orice altă

activitate.

5.6.2 Întreruperi interne

Întreruperile interne provin de la modulele din interiorul microcontrolerului, iar cele mai folosite sunt cele generate de numărătoare. Un numărător poate genera o întrerupere de *depășire* (*overflow*), atunci când valoarea lui trece de maximul reprezentabil și se resetează la zero, sau o întrerupere de *comparare*, atunci când valoarea lui devine egală cu o valoare prestabilită, scrisă într-un registru de comparare. A doua variantă este mai flexibilă, pentru că permite alegerea liberă a perioadei, în timp ce depășirea are întotdeauna perioada dictată de dimensiunea numărătorului și de divizorul de frecvență.

Modulul de conversie analog-numerică generează o întrerupere atunci când o conversie s-a încheiat și rezultatul este disponibil în registrele de date. Aceasta este soluția firească pentru achizițiile de semnal: conversia se pornește, procesorul se ocupă de altceva timp de zeci de microsecunde, iar rutina de tratare preia rezultatul și, eventual, pornește următoarea conversie.

În sfârșit, există surse specifice fiecărei familii de microcontrolere: oscilatorul și detectorul de tensiune scăzută, numărătorul de gardă (*watchdog timer*), care pe ATmega328P poate fi configurat să genereze o întrerupere în loc de o resetare, comparatorul analogic și modulul de scriere în memoria EEPROM, care anunță când operația de scriere, foarte lentă, s-a încheiat.

Observație

Indicatorul de întrerupere se șterge, pe AVR, în două moduri diferite, în funcție de sursă. La majoritatea surselor el este șters automat de hardware în momentul în care se intră în rutina de tratare. Dacă însă tratăm evenimentul prin interogare, fără rutină, indicatorul trebuie șters manual, iar particularitatea este că ștergerea se face scriind valoarea *unu* în bitul respectiv, nu zero. La alte surse, cum este recepția pe USART, indicatorul se șterge de la sine în momentul în care se citește registrul de date; dacă rutina de tratare uită să citească UDR0, indicatorul rămâne setat, iar întreruperea reapare imediat, la nesfârșit.

5.7 Priorități și imbricarea întreruperilor

Prioritatea întreruperilor determină ordinea în care acestea sunt tratate atunci când apar mai multe simultan sau când o cerere nouă sosește în timpul tratării alteia. Ambele situații sunt frecvente în aplicații reale, iar modul în care microcontrolerul le rezolvă diferă semnificativ de la o arhitectură la alta.

5.7.1 Priorități hardware și priorități software

În general, prioritățile pot fi stabilite prin hardware sau prin software. Prioritatea hardware este fixată prin construcția cipului și nu poate fi modificată; prioritatea software poate fi aleasă de programator, prin scrierea unui nivel într-un registru de configurare. Arhitecturile pe treizeci și doi de biți oferă de obicei un controler de întreruperi cu nivele programabile, în care fiecare sursă primește un număr, iar o sursă cu număr mai mic o poate întrerupe pe

una cu număr mai mare.

Un mecanism suplimentar, prezent la toate arhitecturile, este *masca de întrerupere*: posibilitatea de a dezactiva temporar anumite surse, indiferent de prioritatea lor. O sursă mascată nu dispăre, ci doar nu mai este luată în considerare; indicatorul ei continuă să fie setat de hardware, iar în momentul în care masca este ridicată, cererea acumulată este tratată imediat. Masca este instrumentul cu care protejăm secțiunile critice ale programului principal.

Pe arhitectura 8051, foarte răspândită și des citată ca referință, ordinea de prioritate implicită este: întreruperea externă 0, întreruperea de la numărătorul 0, întreruperea externă 1, întreruperea de la numărătorul 1 și, în final, întreruperea de la interfața serială. Prima are prioritatea cea mai mare, ultima cea mai mică, iar programatorul poate ridica o sursă într-un nivel superior prin registrul de prioritate.

5.7.2 Cum stau lucrurile pe AVR

Pe AVR situația este mai simplă și, tocmai de aceea, ușor de înțeles greșit. Prioritatea este stabilită exclusiv de **poziția vectorului în tabel**: o sursă situată mai aproape de adresa zero are prioritate mai mare decât una situată mai departe. Prin urmare, INT0 are prioritate mai mare decât INT1, care are prioritate mai mare decât întreruperile de numărător, care la rândul lor au prioritate mai mare decât cele de la USART sau ADC. Această ordine nu poate fi schimbată prin niciun registru.

Trebuie însă înțeles corect *când* contează această prioritate. Ea intervine numai în situația în care două sau mai multe cereri sunt în așteptare simultan, în momentul în care procesorul este gata să accepte o întrerupere. În acest caz se tratează întâi cea cu vectorul mai mic, iar cealaltă rămâne în așteptare, cu indicatorul setat, și este tratată imediat după. Nicio cerere nu se pierde.

În schimb, pe AVR **nu există preempțiune între întreruperi**. Odată intrată în rutina de tratare, cu bitul I șters automat, nicio altă întrerupere nu o poate întrerupe, oricât de mare i-ar fi prioritatea. O cerere de INT0 sosită în timpul tratării unei întreruperi de la ADC nu îl scoate pe procesor din rutina ADC; ea așteaptă cuminte până la instrucțiunea **reti**. Din acest motiv, pe AVR prioritatea contează mult mai puțin decât durata rutinelor: cea mai bună garanție de răspuns rapid pentru o sursă critică nu este poziția ei în tabel, ci faptul că toate celelalte rutine sunt scurte.

Programatorul poate, teoretic, să reactiveze întreruperile în interiorul unei rutine, executând instrucțiunea **sei()** sau folosind macroul **ISR_NOBLOCK** în locul lui **ISR**. Se obține astfel imbricarea rutinelor: o rutină în curs poate fi întreruptă de alta. Acest lucru este aproape întotdeauna o idee proastă, din trei motive. Primul este consumul de stivă: fiecare nivel de imbricare adaugă adresa de revenire și întregul context salvat, iar pe un microcontroler cu doi kiloocteți de memorie de date stiva se poate întâlni foarte repede cu variabilele. Al doilea este riscul de recursivitate: dacă rutina reactivează întreruperile înainte ca indicatorul propriei surse să fie tratat, sau dacă sursa este activă pe nivel, rutina se poate apela pe sine la nesfârșit. Al treilea, și cel mai insidios, este că orice raționament despre corectitudinea

programului devine mult mai greu, pentru că numărul de întrețeseri posibile ale execuției crește exploziv. Soluția corectă la o rutină care durează prea mult nu este imbricarea, ci scurtarea rutinei.

5.8 Variabile partajate și condiții de cursă

Am stabilit că rutina de tratare comunică cu programul principal exclusiv prin variabile globale. Această comunicare pare banală, dar ascunde două capcane care produc unele dintre cele mai greu de depanat erori din programarea microcontrolerelor.

5.8.1 Optimizarea compilatorului și cuvântul cheie volatile

Prima capcană ține de compilator. Atunci când compilează bucla principală, compilatorul analizează codul și încearcă să îl facă mai rapid. Dacă vede o variabilă globală citită într-o buclă, dar niciodată modificată în interiorul acelei bucle sau în funcțiile apelate din ea, concluzia lui logică este că valoarea nu se schimbă. În consecință, o citește o singură dată, o păstrează într-un registru și folosește registrul în toate iterațiile următoare.

Raționamentul este perfect corect din punctul de vedere al codului vizibil, dar este greșit în realitate, pentru că rutina de tratare modifică variabila fără ca vreun apel să apară în program. Rezultatul este o buclă care așteaptă la nesfârșit un indicator deja setat.

Exemplu 5.3 Variabila partajata intre rutina si programul principal

```

1  volatile uint8_t eveniment = 0;    // fara volatile , bucla nu se
    ↪ termina niciodata
2
3  ISR(INT0_vect)
4  {
5      eveniment = 1;                // rutina doar constata
    ↪ evenimentul
6  }
7
8  int main(void)
9  {
10     configureaza_intreruperea();
11     sei();
12
13     while (1)
14     {
15         if (eveniment)              // programul principal il
    ↪ prelucreaza
16         {
17             eveniment = 0;
18             PORTB ^= (1 << PB5);

```

```

19         }
20     }
21 }

```

Cuvântul cheie **volatile** îi spune compilatorului că variabila se poate schimba din motive pe care el nu le poate vedea și că, prin urmare, fiecare citire din program trebuie să ajungă efectiv la memorie. Regula practică este simplă: orice variabilă globală scrisă într-o rutină de tratare și citită în programul principal, sau invers, trebuie declarată **volatile**. Costul este o mică pierdere de viteză, complet nesemnificativă în comparație cu riscul.

5.8.2 Condiția de cursă

A doua capcană este mai subtilă și nu poate fi rezolvată cu **volatile**. Ea se numește *condiție de cursă* (*race condition*) și apare atunci când corectitudinea rezultatului depinde de momentul exact în care sosește întreruperea.

Procesorul din ATmega328P are registre de opt biți. O variabilă de șaisprezece biți nu poate fi citită printr-o singură instrucțiune; compilatorul generează două citiri succesive, una pentru octetul mai puțin semnificativ și una pentru cel mai semnificativ. Între cele două instrucțiuni există o graniță de instrucțiune, adică exact locul în care o întrerupere poate fi acceptată.

Exemplu 5.4 Citire neatomică a unei variabile pe 16 biți

```

1  volatile uint16_t impulsuri = 0;
2
3  ISR(INT0_vect)
4  {
5      impulsuri++;           // incrementat la fiecare impuls de la
                             ↪ senzor
6  }
7
8  int main(void)
9  {
10     uint16_t copie;
11     // ...
12     while (1)
13     {
14         copie = impulsuri;   // DOUA citiri de octet, nu una
                             ↪ singura
15         afiseaza(copie);
16     }
17 }

```

Să urmărim ce se poate întâmpla. Presupunem că variabila `impulsuri` are valoarea 255 =

0x00FF. Programul principal citește octetul inferior și obține 0xFF. Chiar în acest moment sosește un impuls de la senzor, rutina de tratare se execută și incrementează variabila, care devine $256 = 0x0100$. Programul principal își reia execuția și citește octetul superior, obținând acum 0x01. Valoarea recompusă este $0x01FF = 511$.

Numărul 511 nu a existat niciodată. Variabila a avut valoarea 255, apoi valoarea 256, dar niciodată 511. Programul a citit jumătate dintr-o valoare veche și jumătate dintr-una nouă, iar rezultatul este o eroare de peste o sută la sută, apărută o dată la câteva mii de citiri, adică exact genul de defect care trece de toate testele și se manifestă la client.

Soluția este ca citirea să devină *atomică*, adică să nu poată fi întreruptă. Cea mai directă metodă este dezactivarea temporară a întreruperilor pe durata citirii, cu grija de a restaura starea anterioară și nu de a le reactiva necondiționat: dacă funcția care conține citirea este apelată dintr-un context în care întreruperile erau deja oprite, un `sei()` necondiționat le-ar reporni prematur.

Exemplu 5.5 Secțiune critică protejată corect

```
1  #include <util/atomic.h>
2
3  uint16_t citeste_impulsuri(void)
4  {
5      uint16_t copie;
6
7      ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
8      {
9          copie = impulsuri;          // cele doua citiri nu pot fi
                                     ↪ intrerupte
10     }
11     return copie;
12 }
13
14 // varianta echivalenta, scrisa manual:
15 uint16_t citeste_impulsuri_manual(void)
16 {
17     uint8_t sreg_salvat = SREG;      // retine starea bitului I
18     uint16_t copie;
19
20     cli();                          // opreste intreruperile
21     copie = impulsuri;
22     SREG = sreg_salvat;              // restaureaza starea, nu
                                     ↪ forteaza sei()
23     return copie;
24 }
```

Secvența de instrucțiuni protejată în acest fel se numește *secțiune critică*. Ea trebuie să fie cât mai scurtă posibil, pentru că pe durata ei sistemul este surd la orice eveniment; o secțiune critică lungă produce exact aceleași probleme ca o rutină de tratare lungă. Se protejează accesul, nu prelucrarea: copiem valoarea în interiorul secțiunii critice și o prelucrăm după ieșirea din ea.

Observație

Nu orice acces are nevoie de protecție. Pe AVR, citirea și scrierea unei variabile de opt biți sunt operații atomice prin construcție, pentru că se execută într-o singură instrucțiune. O variabilă de tip `uint8_t` folosită ca simplu indicator poate fi deci citită fără secțiune critică. În schimb, o operație de tipul `contor++` nu este atomică nici măcar pe opt biți, pentru că se traduce în citire, incrementare și scriere, adică trei instrucțiuni separate. Dacă atât rutina de tratare, cât și programul principal incrementează aceeași variabilă, incrementele se pot pierde.

5.9 Aplicație cu sistemul de întreruperi

Vom aplica acum tot ce am discutat la o problemă concretă, aceeași care apare în orice introducere în subiect: comanda unui LED de la un buton, folosind sistemul de întreruperi externe. Cerința este ca LED-ul să comute starea, adică să se aprindă dacă era stins și invers, la fiecare acționare a butonului, iar comutarea să se producă pe frontul crescător al semnalului.

5.9.1 Montajul

Butonul se conectează la pinul PD2 al microcontrolerului, singurul care poate genera întreruperea externă INT0. Pentru ca frontul să fie crescător, butonul se leagă între pin și tensiunea de alimentare, iar pinul este menținut la masă printr-o rezistență de coborâre (*pull-down*) de ordinul a 10 k Ω . Astfel, în repaus intrarea este la nivel jos, iar la apăsare trece la nivel înalt, producând frontul crescător cerut.

LED-ul se conectează la pinul PB5, în serie cu o rezistență de limitare a curentului. Valoarea acesteia se calculează din tensiunea de alimentare, tensiunea directă a LED-ului și curentul dorit, respectând limita de curent pe pin dată în fișa de catalog.

Alternativa, foarte răspândită în practică, este legarea butonului între pin și masă, cu activarea rezistenței interne de ridicare (*pull-up*) prin scrierea unui unu în bitul corespunzător din PORTD. În acest caz apăsarea produce un front căzător, iar configurarea întreruperii trebuie schimbată în consecință. Varianta cu rezistență internă are avantajul că elimină o componentă externă.

5.9.2 Inițializarea porturilor

Prima etapă a programului este configurarea direcțiilor pinilor. Pinul LED-ului trebuie să fie ieșire, iar pinul butonului intrare. Scrierea explicită a fiecărui bit, așa cum o generează majoritatea instrumentelor de configurare automată, are avantajul de a documenta intenția

pentru toți pinii portului.

Exemplu 5.6 Initializarea porturilor pentru aplicatia propusa

```

1  #include <avr/io.h>
2  #include <avr/interrupt.h>
3
4  static void init_porturi(void)
5  {
6      // Portul B: PB5 iesire (LED), restul intrari
7      DDRB = (1 << DDB5);
8      PORTB = 0x00;           // LED stins la pornire
9
10     // Portul D: PD2 intrare (buton INT0), restul intrari
11     DDRD = 0x00;
12     PORTD = 0x00;           // fara pull-up intern;
                             ↪ rezistenta este externa
13 }
```

5.9.3 Inițializarea sistemului de întreruperi

A doua etapă este configurarea propriu-zisă a întreruperii externe. Sunt necesare trei operații distincte, în ordinea aceasta: alegerea modului de declanșare, ștergerea unui eventual indicator rămas din trecut și activarea locală a sursei.

Modul de declanșare pentru INT0 se scrie în registrul EICRA, prin biții ISC01 și ISC00. Pentru frontul crescător ambii biți trebuie setați. Ștergerea indicatorului INTF0 din registrul EIFR se face, așa cum am arătat mai devreme, scriind unu în poziția respectivă; operația este necesară pentru că simpla schimbare a modului de declanșare poate seta indicatorul, iar fără această curățare rutina s-ar executa o dată nejustificat imediat după activarea globală. Activarea locală se face prin bitul INT0 din registrul EIMSK.

Exemplu 5.7 Configurarea intreruperii externe INT0 pe front crescator

```

1  static void init_intrerupere(void)
2  {
3      // ISC01 = 1, ISC00 = 1 → front crescator pe INT0
4      EICRA |= (1 << ISC01) | (1 << ISC00);
5
6      // sterge un eventual indicator ramas (se scrie 1, nu 0)
7      EIFR |= (1 << INTF0);
8
9      // activare locala a sursei INT0
10     EIMSK |= (1 << INT0);
11 }
```

Merită subliniat că numele INTO apare aici în două roluri diferite: ca nume al sursei de întrerupere și ca nume al bitului din EIMSK. Este o convenție comodă, dar care poate deruta la prima citire a fișei de catalog.

5.9.4 Rutina de tratare și activarea globală

Rutina de tratare conține o singură operație: comutarea bitului corespunzător LED-ului prin operatorul sau-exclusiv, aplicat între registrul PORTB și masca bitului 5. Această scriere inversează bitul 5 și lasă neschimbați ceilalți biți ai portului, ceea ce este exact comportamentul dorit.

Exemplu 5.8 Rutina de tratare si bucla principala

```

1 ISR(INT0_vect)
2 {
3     PORTB ^= (1 << PB5);      // comuta LED-ul; nimic altceva
4 }
5
6 int main(void)
7 {
8     init_porturi();
9     init_intrerupere();
10
11     sei();                    // activare globala: SREG |= (1 <<
                               ↪ 7)
12
13     while (1)
14     {
15         // programul principal poate face orice altceva;
16         // butonul este tratat independent de ce se intampla aici
17     }
18     return 0;
19 }

```

Ultimul pas, activarea globală prin `sei()`, se face întotdeauna la sfârșitul inițializării, după ce toate perifericele au fost configurate. Motivul este simplu: dacă întreruperile ar fi activate mai devreme, o rutină de tratare s-ar putea executa înainte ca structurile de date pe care le folosește să fi fost inițializate. Simetric, la oprirea controlată a unui subsistem se dezactivează întâi întreruperea și abia apoi se demontează structurile.

Se observă cât de goală este bucla principală. Aceasta este imaginea corectă a unui program bazat pe întreruperi: comportamentul reactiv este descris în rutine, iar bucla principală rămâne liberă pentru prelucrări sau, într-o aplicație alimentată de la baterie, pentru trecerea procesorului într-un regim de consum redus din care întreruperea îl va trezi.

5.9.5 Varianta Arduino

Mediul Arduino oferă o abstractizare peste același mecanism, prin funcția `attachInterrupt()`. Ea primește numărul intern al întreruperii, obținut din numărul pinului cu ajutorul macroului `digitalPinToInterrupt()`, adresa funcției care va fi executată și modul de declanșare, exprimat prin constantele `RISING`, `FALLING`, `CHANGE` sau `LOW`.

Exemplu 5.9 Aceeași aplicație, scrisă cu funcțiile Arduino

```

1  const uint8_t PIN_BUTTON = 2;      // PD2, corespunde lui INT0
2  const uint8_t PIN_LED     = 13;     // PB5, LED-ul de pe placa
3
4  volatile bool cerere_comutare = false;
5
6  void trateaza_buton()                // rutina de tratare
7  {
8      cerere_comutare = true;        // doar marcheaza evenimentul
9  }
10
11 void setup()
12 {
13     pinMode(PIN_LED, OUTPUT);
14     pinMode(PIN_BUTTON, INPUT);     // rezistenta de coborare este
                                     ↪ externa
15
16     attachInterrupt(digitalPinToInterrupt(PIN_BUTTON),
17                     trateaza_buton,
18                     RISING);
19 }
20
21 void loop()
22 {
23     if (cerere_comutare)
24     {
25         cerere_comutare = false;
26         digitalWrite(PIN_LED, !digitalRead(PIN_LED));
27     }
28 }
```

Diferența față de varianta cu registre nu este doar de sintaxă. Aici rutina de tratare nu mai comandă direct ieșirea, ci doar ridică un indicator, iar comutarea propriu-zisă se face în `loop()`. Motivul este că funcțiile `digitalRead()` și `digitalWrite()` din Arduino nu sunt operații elementare, ci funcții care traduc numărul pinului în port și mască, verificând pe

parcurs dacă pinul respectiv nu este cumva folosit de modulul de modulare în durată. Ele durează câteva microsecunde, ceea ce este mult pentru o rutină de tratare. Am aplicat deci exact tiparul recomandat mai devreme: rutina constată, bucla prelucrează.

Se observă și obligativitatea lui `volatile` pe variabila `cerere_comutare`: fără el, compilatorul ar avea tot dreptul să considere că valoarea citită în `loop()` nu se schimbă niciodată.

5.9.6 Problema oscilațiilor de contact

Programul de mai sus funcționează la prima încercare doar în simulare. Pe un montaj real, LED-ul va comuta uneori o dată, alteori de trei sau de cinci ori la o singură apăsare. Cauza este fenomenul de *oscilație a contactelor* (*contact bounce*): lamelele metalice ale butonului nu se lipesc instantaneu, ci se ciocnesc și se desprind de câteva ori pe parcursul unui interval de ordinul milisecundelor, generând o serie de fronturi în loc de unul singur. Sistemul de întreruperi, spre deosebire de ochiul uman, le vede pe toate.

Soluția clasică în software este de a ignora evenimentele care sosesc prea repede după unul deja acceptat. Se reține momentul ultimei acceptări și se compară cu momentul curent, iar dacă diferența este mai mică decât o durată aleasă, de obicei între douăzeci și cincizeci de milisecunde, evenimentul se consideră ecou și se abandonează.

Exemplu 5.10 Filtrarea oscilațiilor de contact în rutina de tratare

```

1  const unsigned long INTERVAL_MIN = 40;    // milisecunde
2
3  volatile bool        cerere_comutare = false;
4  volatile unsigned long ultimul_moment  = 0;
5
6  void trateaza_buton()
7  {
8      unsigned long acum = millis();          // valoarea poate fi
          ↪ citita , dar nu avanseaza
9
10     if (acum - ultimul_moment >= INTERVAL_MIN)
11     {
12         cerere_comutare = true;
13         ultimul_moment  = acum;
14     }
15 }
```

Comparația este scrisă intenționat sub forma „diferența dintre momentul curent și cel reținut este cel puțin egală cu intervalul minim” și nu sub forma „momentul curent este cel puțin egal cu momentul reținut plus intervalul minim”. Prima formă rămâne corectă și atunci când numărătorul de milisecunde depășește valoarea maximă și revine la zero, pentru că scăderea în aritmetica fără semn dă oricum diferența corectă; a doua formă produce, la momentul depășirii, un buton care nu mai răspunde pentru un interval lung de timp.

Există și soluția hardware, un filtru trece-jos format dintr-o rezistență și un condensator, eventual urmat de un trigger Schmitt. Ea are avantajul că nu consumă timp de procesor și dezavantajul unei componente în plus pentru fiecare buton.

Întrebări de verificare

1. Explicați de ce interogarea ciclică nu poate garanta detecția unui eveniment scurt, chiar dacă bucla principală este optimizată. Ce mărime a programului trebuie comparată cu durata evenimentului?
2. Enumerați, în ordine, operațiile pe care le execută procesorul AVR între momentul acceptării unei cereri de întrerupere și prima instrucțiune a rutinei de tratare. Precizați care dintre ele sunt realizate de hardware și care de codul generat de compilator.
3. De ce procesorul termină întotdeauna instrucțiunea aflată în curs de execuție înainte de a trata o întrerupere? Ce s-ar putea întâmpla dacă ar întrerupe la mijlocul unei instrucțiuni de scriere în memorie?
4. Care este diferența dintre instrucțiunile **ret** și **reti**? Descrieți comportamentul unui program în care o rutină de tratare s-ar încheia din greșeală cu **ret**.
5. Ce se înțelege prin contextul procesorului și de ce trebuie el salvat la intrarea în rutina de tratare? Dați un exemplu concret de eroare produsă de nesalvarea registrului de stare **SREG**.
6. Un microcontroler ATmega328P se resetează la intervale neregulate, deși programul nu conține nicio instrucțiune de resetare și numărătorul de gardă este dezactivat. Formulați o explicație plauzibilă, pornind de la structura tabelului vectorilor de întrerupere.
7. Explicați de ce, pe ATmega328P, fiecare intrare din tabelul vectorilor ocupă două cuvinte, iar pe un microcontroler cu memorie de program mică ar putea ocupa doar unul. Ce instrucțiune este plasată în fiecare caz?
8. Pe AVR, INT0 are prioritate mai mare decât întreruperea de recepție a interfeței USART. O cerere INT0 sosește în timp ce se execută rutina de tratare a recepției. Ce se întâmplă și de ce? În ce situație ar conta totuși diferența de prioritate?
9. Argumentați de ce reactivarea întreruperilor în interiorul unei rutine de tratare este, în general, o practică nerecomandată. Care sunt riscurile pentru memoria de date?
10. O variabilă globală de opt biți este scrisă într-o rutină de tratare și citită în bucla principală, fără să fie declarată **volatile**. Descrieți raționamentul compilatorului care duce la un program blocat și explicați ce anume schimbă declarația **volatile**.

11. Un numărător de impulsuri pe şaisprezece biţi este incrementat într-o rutină de tratare şi citit în programul principal. Construiţi o secvenţă de evenimente în care programul principal obţine o valoare care nu a existat niciodată şi indicaţi cele două soluţii posibile pentru citirea corectă.
12. Un buton legat la INT0 comută uneori de mai multe ori la o singură apăsare. Explicaţi fenomenul fizic responsabil şi comparaţi soluţia software cu cea hardware, din punctul de vedere al timpului de procesor consumat şi al numărului de componente.

6. Module de tip timer

Prezentare generală
Clasificarea modulelor de tip timer
Modalități de accesare a datelor din timer
Utilizarea timerului cu întrerupere
Circuitul de prescalare
Caracteristicile semnalelor periodice
Generarea unui semnal cu factor de umplere variabil (PWM)
Implementarea PWM
Capturarea evenimentelor externe
Concluzii
Întrebări de verificare

6.1 Prezentare generală

Până în acest punct al cursului, microcontrolerul a fost privit ca o mașină care execută instrucțiuni una după alta: citește o intrare, ia o decizie, comandă o ieșire. Într-un asemenea model, timpul nu există ca mărime de sine stătătoare; el apare doar indirect, ca durata pe care o consumă instrucțiunile executate. Aproape orice aplicație reală are însă nevoie de timp ca de o mărime pe care să o poată măsura și impune: un semafor trebuie să țină verde exact treizeci de secunde, un motor trebuie alimentat cu o putere medie reglabilă, un senzor cu ultrasunete trebuie să spună cât de lat a fost un impuls cu o precizie de câteva microsecunde, iar un protocol de comunicație trebuie să eșantioneze linia exact la mijlocul fiecărui bit. Pentru toate acestea, microcontrolerul dispune de un bloc hardware dedicat: *modulul de tip timer*.

Un timer este, la bază, un lucru foarte simplu: un *numărător* binar implementat în hardware, adică un registru care își crește valoarea cu o unitate la fiecare impuls primit pe intrarea sa de tact. Ceea ce îl face util nu este complexitatea lui, ci faptul că numără *singur*, fără ca procesorul să execute vreo instrucțiune. Odată pornit, numărătorul avansează în paralel cu programul, indiferent dacă unitatea centrală calculează, așteaptă, execută o rutină de întrerupere sau este în regim de consum redus. Din acest motiv, un timer este una dintre puținele periferice care oferă o măsură a timpului independentă de ceea ce face software-ul.

Impulsurile pe care le numără un timer provin, în cazul obișnuit, din ceasul sistemului, adică din oscilatorul care ritmează întregul microcontroler. Pe o placă Arduino Uno acest

oscilator este un rezonator ceramic sau un cuarț de 16 MHz, deci un impuls la fiecare 62,5 ns. Frecvența ceasului fiind cunoscută și stabilă, numărul de impulsuri acumulate în registrul numărătorului se traduce direct într-un interval de timp: dacă numărătorul a avansat cu 1600 de unități, au trecut $1600 \times 62,5 \text{ ns}$, adică $100 \mu\text{s}$. Aceasta este întreaga idee a temporizării în microcontrolere - transformarea unei durate într-un număr.

6.1.1 Timer sau counter

Distincția între cele două denumiri, folosite adesea alternativ, ține de natura semnalului numărat. Vorbim despre un *timer* atunci când semnalul aplicat la intrarea de tact are o distribuție în timp constantă și uniformă, adică este un semnal de ceas cu perioadă cunoscută. În acest caz, numărul acumulat este proporțional cu timpul scurs, iar modulul măsoară durate. Vorbim despre un *counter* (numărător de evenimente) atunci când semnalul aplicat provine din exterior și are o distribuție aleatoare în timp: impulsuri de la un encoder rotativ, de la un debitmetru cu turbină, de la o barieră optică. În acest al doilea caz, numărul acumulat nu mai spune nimic despre timp, ci despre câte evenimente s-au produs.

Hardware-ul este însă același. Pe ATmega328P, aceleași module pot fi comutate prin configurare între cele două regimuri: dacă biții de selecție a sursei de tact indică ceasul intern, modulul lucrează ca timer; dacă indică pinul extern T0 sau T1, modulul lucrează ca numărător de impulsuri externe, sensibil pe front crescător sau descrescător, după cum s-a configurat. Această dualitate explică denumirea oficială din foaia de catalog, *Timer/Counter*.

6.1.2 La ce folosește un timer

Utilizările unui asemenea modul acoperă o gamă largă, dar toate se reduc la câteva mecanisme de bază. Cronometrarea este cea mai directă: numărătorul este pornit la începutul unui interval și citit la sfârșit, diferența dintre cele două valori dând durata. Generarea de întreruperi periodice este poate cea mai folosită în practică: modulul este configurat astfel încât să semnaleze procesorului, la intervale exacte, că a trecut o unitate de timp, iar programul își organizează activitatea în jurul acestor semnale. Generarea de semnale periodice la un pin permite obținerea, fără intervenția procesorului, a unui semnal dreptunghiular cu frecvență și factor de umplere impuse, folosit la comanda motoarelor, a iluminării sau la sinteza de sunet. Măsurarea frecvenței sau a lățimii unui impuls extern se face fie numărând evenimente într-o fereastră de timp fixă, fie memorând valoarea numărătorului în momentul apariției unui front.

Ceea ce leagă toate aceste utilizări este o economie de resurse. Un microcontroler care nu are timere poate, teoretic, să facă aceleași lucruri prin software, numărând instrucțiuni; dar în acel moment procesorul este ocupat integral cu numărarea și nu mai poate face altceva, iar orice întrerupere care survine falsifică rezultatul. Timerul mută sarcina din software în hardware, ceea ce are trei consecințe imediate: procesorul rămâne liber pentru restul aplicației, precizia nu mai depinde de calea parcursă prin program, iar comportarea aplicației devine reproductibilă.

6.1.3 De ce funcția `delay()` este cea mai proastă soluție

Orice student care a scris un program de intermitență pentru un LED cunoaște funcția `delay()`. Ea este simplă, se învață în două minute și funcționează la prima încercare. Este, în același timp, cea mai proastă modalitate de a măsura timpul dintre cele existente, iar motivul merită înțeles în detaliu, pentru că toată materia acestui capitol este, într-un fel, răspunsul la această problemă.

Funcția `delay(1000)` nu face nimic timp de o secundă. Nu este o figură de stil: intern, funcția execută o buclă care citește repetat contorul de milisecunde întreținut de bibliotecă și iese din buclă atunci când acesta a crescut cu o mie. În tot acest răstimp, procesorul consumă cicluri mașină pentru a compara două numere. Un ATmega328P la 16 MHz execută aproximativ șaisprezece milioane de instrucțiuni simple pe secundă; o secundă de `delay()` înseamnă, în ordin de mărime, milioane de instrucțiuni cheltuite pentru a nu face nimic. Această risipă nu ar fi gravă în sine, dacă nu ar avea o consecință mult mai serioasă: cât timp programul așteaptă în `delay()`, el nu poate răspunde la nimic altceva. Un buton apăsat și eliberat în acest interval nu va fi observat. Un caracter sosit pe interfața serială va fi pierdut dacă tamponul se umple. O comandă de oprire de urgență va fi luată în seamă abia peste o secundă.

Situația se agravează atunci când aplicația trebuie să facă două lucruri cu ritmuri diferite - să clipească un LED la fiecare 500 ms și să citească un senzor la fiecare 200 ms. Cu `delay()`, cele două cerințe nu se pot împăca decât prin calcule de intervale comune, iar programul devine repede de neîntreținut. Se adaugă și o eroare sistematică: timpul consumat de instrucțiunile dintre două apeluri `delay()` se adună la întârziere, astfel încât perioada reală este întotdeauna mai mare decât cea cerută, cu o cantitate care depinde de ramura parcursă prin program.

Prima ameliorare, cunoscută sub numele de temporizare neblokantă, înlocuiește așteptarea cu o comparație de momente de timp. Programul nu mai stă pe loc, ci verifică la fiecare trecere prin bucla principală dacă a sosit momentul de a acționa.

Exemplu 6.1 Temporizare blocantă și echivalentul ei neblokant

```
1 // Varianta blocantă: procesorul este imobilizat 500 ms
2 void loop() {
3     digitalWrite(LED, HIGH);
4     delay(500);
5     digitalWrite(LED, LOW);
6     delay(500);
7 }
8
9 // Varianta neblokantă: bucla se parcurge de mii de ori pe secunda
10 unsigned long ultimaComutare = 0;
11 const unsigned long PERIOADA = 500;
12
```

```

13 void loop() {
14     unsigned long acum = millis();
15     if (acum - ultimaComutare >= PERIOADA) {
16         ultimaComutare = acum;           // se retine momentul, nu se
            ↪ aduna
17         stareLed = !stareLed;
18         digitalWrite(LED, stareLed);
19     }
20     // aici pot fi tratate butoane, senzori, comunicatie seriala
21 }

```

Este important de observat că nici această variantă nu ar fi posibilă fără un timer: funcția `millis()` returnează un contor de milisecunde care este incrementat de rutina de întrerupere a unui timer hardware. Diferența este că acum programul *consultă* timpul în loc să îl *aștepte*. Etapa următoare, tratată în acest capitol, este să lăsăm timerul să anunțe singur programul, prin întrerupere, sau chiar să genereze semnalul dorit direct la pin, fără ca procesorul să fie implicat deloc.

6.2 Clasificarea modulelor de tip timer

Sub denumirea generică de modul de timp se ascund mai multe structuri, care diferă prin sursa impulsurilor numărate, prin ceea ce se întâmplă când numărătorul ajunge la capăt și prin legătura cu pinii de ieșire. O clasificare utilă, orientată către funcție, distinge timerul simplu, numărătorul de evenimente, generatorul de PWM, temporizatorul de gardă și ceasul de timp real.

6.2.1 Timerul simplu

Timerul simplu numără impulsurile ceasului intern și își incrementează registrul de contor la fiecare front activ, crescător sau descrescător, după configurare. Când valoarea atinge maximul reprezentabil, numărătorul se resetează la zero - operație numită *depășire* sau *overflow* - și poate semnala acest lucru printr-un indicator de stare și printr-o întrerupere. Unele module pot număra și descrescător, iar altele pot alterna cele două sensuri.

Domeniul de numărare este determinat de numărul de biți ai registrului de contor. Pentru o rezoluție de n biți, numărătorul parcurge intervalul

$$[0, 2^n - 1],$$

deci trece prin 2^n stări distincte înainte de a reveni la zero. În practică, modulele de timp sunt de 8 sau de 16 biți, ceea ce dă intervalele $[0, 255]$, respectiv $[0, 65535]$. Această cifră aparent banală are consecințe directe asupra a ceea ce se poate măsura: un numărător de 8 biți alimentat direct cu ceasul de 16 MHz se umple în $256 \times 62,5$ ns, adică în $16 \mu\text{s}$, în timp ce unul de 16 biți, în aceleași condiții, rezistă $65536 \times 62,5$ ns, adică $4,096$ ms. Nici una

dintre aceste durate nu este suficientă pentru a cronometra o secundă, motiv pentru care apare necesitatea circuitului de prescalare, discutat mai departe.

6.2.2 Numărătorul de evenimente

Numărătorul propriu-zis este același modul, cu deosebirea că intrarea de tact este comutată pe un pin extern. Fiecare tranziție a semnalului aplicat pe acel pin, în sensul configurat, incrementează registrul de contor. Modulul nu mai măsoară timp, ci acumulează evenimente: rotații ale unui arbore, impulsuri de la un contor de apă, treceri prin dreptul unui senzor optic. Combinația celor două regimuri este deosebit de utilă - un timer măsoară o fereastră de exact o secundă, iar un numărător acumulează în acest interval impulsurile externe; valoarea citită la sfârșit este chiar frecvența semnalului, exprimată în hertzi.

6.2.3 Timerul generator de PWM

Un modul configurat în regim PWM produce un semnal digital cu frecvență constantă și cu *factor de umplere* variabil, adică un semnal la care raportul dintre durata stării active și perioadă poate fi reglat prin software. Generarea are loc integral în hardware, prin compararea continuă a valorii numărătorului cu o valoare de prag înscrisă într-un registru de comparare. Odată configurat, modulul menține semnalul la pin nedefinit de mult timp, fără nicio instrucțiune executată de procesor; programul intervine doar atunci când dorește să schimbe factorul de umplere, scriind o singură valoare într-un registru. Aplicațiile tipice sunt comanda motoarelor de curent continuu, reglarea intensității luminoase a LED-urilor, sinteza de semnale audio simple și realizarea unui convertor numeric-analogic elementar prin filtrarea semnalului.

6.2.4 Temporizatorul de gardă

Watchdog timer-ul este un temporizator specializat, a cărui unică misiune este să repornească microcontrolerul atunci când programul a încetat să funcționeze corect. Mecanismul este simplu și eficient: modulul numără continuu, independent de programul principal, iar programul are obligația de a-l reinițializa periodic, printr-o instrucțiune dedicată. Dacă programul se blochează într-o buclă infinită, dacă intră într-o rutină de întrerupere din care nu mai iese sau dacă o perturbăție electromagnetică a alterat conținutul contorului de program, reinițializarea nu mai are loc, temporizatorul ajunge la capăt și generează un semnal de reset care repornește sistemul într-o stare cunoscută. Perioada se alege prin configurare, de la câteva milisecunde la câteva secunde, în funcție de ritmul buclei principale. În aplicațiile critice, unde un blocaj nu poate fi remediat de un operator care apasă un buton, prezența unui watchdog este obligatorie.

6.2.5 Ceasul de timp real

Real-Time Clock-ul, prescurtat RTC, este un modul destinat păstrării orei și datei calendaristice pe termen lung. Spre deosebire de un timer obișnuit, el nu se resetează la fiecare depășire, ci întreține o structură de timp continuă, și continuă să funcționeze în regim de

consum redus, adesea alimentat dintr-o baterie separată atunci când restul sistemului este oprit. Sursa lui de tact este de regulă un cuarț de 32,768 Hz; valoarea nu este întâmplătoare, fiind 2^{15} , astfel încât o divizare succesivă cu doi, realizată de un lanț de cincisprezece bistabile, produce exact un impuls pe secundă, fără nicio eroare de rotunjire. Un RTC poate genera întreruperi periodice și alarme la ore sau date prestabilite. ATmega328P nu are un RTC complet, dar poate apropia funcția conectând un cuarț de ceasornic la Timer2 în regim asincron; pentru aplicații serioase se folosește însă un circuit extern dedicat.

6.2.6 Timerele microcontrolerului ATmega328P

Microcontrolerul de referință al acestui curs dispune de trei module de timp, prezentate sintetic în Tabelul 6.1. Deosebirile dintre ele nu sunt doar de rezoluție, ci și de dotare: numai Timer1 are unitate de captură a intrării și registre de comparare pe 16 biți, ceea ce îl face singurul potrivit pentru măsurători fine de timp; Timer2 are un set mai bogat de factori de prescalare și poate fi comandat de un cuarț extern de ceasornic.

Tabela 6.1 Modulele de timp ale microcontrolerului ATmega328P

Modul	Rezoluție	Factori de prescalare	Funcții de bibliotecă dependente
Timer0	8 biți	1, 8, 64, 256, 1024	<code>millis()</code> , <code>micros()</code> , <code>delay()</code> , <code>analogWrite()</code> pe pinii 5 și 6
Timer1	16 biți	1, 8, 64, 256, 1024	biblioteca <code>Servo</code> , <code>analogWrite()</code> pe pinii 9 și 10
Timer2	8 biți	1, 8, 32, 64, 128, 256, 1024	<code>tone()</code> , <code>analogWrite()</code> pe pinii 3 și 11

Ultima coloană a tabelului merită citită cu atenție, pentru că explică o categorie întreagă de erori greu de diagnosticat. Funcțiile de bibliotecă Arduino nu sunt magice: ele sunt construite peste aceleași trei module hardware, iar dacă un program reconfigurează un modul, funcțiile care depindeau de el încetează să funcționeze corect. Cazul cel mai supărător este Timer0. Nucleul Arduino îl configurează la inițializare cu factorul de prescalare 64 și îi activează întreruperea de depășire; rutina de tratare a acestei întreruperi incrementează contorul de milisecunde pe care se sprijină `millis()`, `micros()` și `delay()`. Dacă programul schimbă factorul de prescalare al lui Timer0 pentru a obține un PWM de altă frecvență pe pinul 5, va obține într-adevăr acea frecvență, dar în același timp va accelera sau va încetini contorul de milisecunde, iar toate temporizările bazate pe `millis()` vor deveni false, fără niciun mesaj de eroare. Regula practică este simplă: pe o placă Arduino, Timer0 se lasă în pace, iar experimentele de configurare manuală se fac pe Timer1 sau Timer2.

Observație

Reconfigurarea unui timer nu produce niciodată o eroare de compilare și nici un mesaj la execuție. Efectul apare indirect, sub forma unor temporizări care nu mai sunt corecte, a unui servomotor care tremură sau a unei comunicații seriale care pierde caractere. Înainte de a modifica registrele unui timer, verificați ce funcții de bibliotecă îl folosesc.

6.2.7 Modurile de lucru

Comportamentul numărătorului între două reveniri la zero definește *modul de lucru* al modulului. ATmega328P oferă patru familii de moduri, selectate prin biții WGM distribuiți în registrele de control TCCRnA și TCCRnB. Diferența dintre ele privește valoarea maximă la care ajunge numărătorul, numită *TOP*, sensul numărării și momentul în care se comută ieșirea.

În **modul normal**, numărătorul crește de la zero până la valoarea maximă permisă de rezoluție, $2^n - 1$, după care revine la zero și marchează indicatorul de depășire. Perioada evenimentelor este fixă, determinată numai de frecvența de tact și de rezoluție; singurul mod de a o schimba este alegerea factorului de prescalare. Modul normal este potrivit pentru cronometrare generală și pentru numărarea evenimentelor externe, dar este stângaci atunci când se cere un interval anume, pentru că singura soluție este preîncărcarea numărătorului cu o valoare inițială, operație care trebuie repetată la fiecare depășire, în rutina de întrerupere, și care introduce o eroare dependentă de latența acesteia.

În **modul CTC** (*Clear Timer on Compare Match*), numărătorul crește de la zero până când valoarea sa devine egală cu conținutul registrului de comparare OCRnA, moment în care este șters automat, în hardware, și reia numărarea. Rolul de TOP este preluat de registrul de comparare, deci perioada devine programabilă cu pas de un impuls de tact. Ștergerea fiind făcută de hardware, ea nu depinde de momentul în care procesorul ajunge să trateze întreruperea, astfel încât nu se acumulează erori de la o perioadă la alta. Acesta este modul potrivit pentru generarea de întreruperi periodice exacte și pentru obținerea unui semnal dreptunghiular simetric la pin, prin comutarea ieșirii la fiecare egalitate.

În **modul PWM rapid** (*Fast PWM*), numărătorul parcurge continuu domeniul de la zero la TOP, iar ieșirea asociată este pusă în starea activă la începutul ciclului și trecută în starea inactivă atunci când numărătorul atinge valoarea din registrul de comparare. Rezultă un semnal dreptunghiular de frecvență constantă, al cărui factor de umplere este dat de raportul dintre valoarea de comparare și TOP. Deoarece numărătorul parcurge domeniul o singură dată pe perioadă, frecvența obținută este cea mai mare posibilă pentru o rezoluție dată, ceea ce este avantajos atunci când se dorește un filtraj ușor sau un zgomot acustic în afara domeniului audibil.

În **modul PWM cu fază corectă** (*Phase Correct PWM*), numărătorul crește de la zero la TOP, apoi scade înapoi la zero, iar ieșirea își schimbă starea la fiecare trecere prin valoarea de comparare, o dată pe rampa ascendentă și o dată pe cea descendentă. Impulsul rezultat este centrat pe mijlocul perioadei, iar centrul lui rămâne fix indiferent de factorul de umplere. De aici și numele: faza semnalului nu se deplasează atunci când se modifică

lăţimea impulsului. Preţul plătit este că perioada devine aproximativ dublă, deci frecvenţa se înjumătăţeşte faţă de PWM rapid la aceeaşi rezoluţie şi acelaşi factor de prescalare. Modul se foloseşte acolo unde deplasarea fazei ar produce efecte nedorite: comanda motoarelor fără perii, punţile în H care comandă înfăşurări, aplicaţiile audio.

Figura 6.1 arată diferenţa dintre cele două regimuri PWM, prin compararea evoluţiei număratorului şi a semnalului rezultat la ieşire.

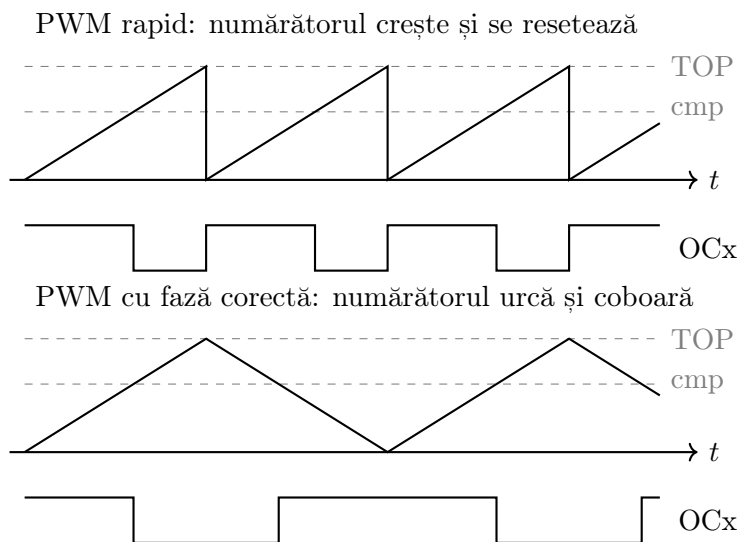


Figura 6.1 Evoluția număratorului și semnalul de ieșire în modul PWM rapid (sus) și în modul PWM cu fază corectă (jos), pentru aceeași valoare de comparare.

6.3 Modalități de accesare a datelor din timer

Un timer care numără în tăcere nu este de niciun folos dacă programul nu poate afla ce valoare a atins sau când s-a produs un eveniment. Există trei căi prin care software-ul intră în contact cu modulul de timp, iar alegerea între ele determină în bună măsură arhitectura aplicației.

6.3.1 Citirea directă a registrului de contor

Cea mai simplă modalitate este citirea valorii curente a număratorului, care este accesibilă ca un registru obișnuit, mapat în spațiul de adrese de intrare-ieșire:

Exemplu 6.2 Citirea directă a registrului de contor

```
1 uint8_t  valoare8  = TCNT0;    // numărator de 8 biti al Timer0
2 uint16_t valoare16 = TCNT1;    // numărator de 16 biti al Timer1
```

Această citire nu oprește numărătorul și nu îl perturbă în niciun fel; ea returnează pur și simplu fotografia valorii din acel moment. Metoda este utilă pentru măsurarea duratei unei secvențe de cod, prin citirea număratorului înainte și după, sau pentru cronometrări scurte, în care nu se produce nicio depășire.

Citirea unui numărator de 16 biți pe o mașină de 8 biți ridică o problemă subtilă. Registrul TCNT1 nu poate fi adus în procesor printr-o singură instrucțiune, ci prin două citiri succesive, ale octetului inferior și ale celui superior. Dacă între cele două citiri număratorul trece de la valoarea 0x00FF la 0x0100, rezultatul recompus poate fi complet greșit. Hardware-ul rezolvă parțial problema printr-un registru temporar: la citirea octetului inferior, octetul superior este copiat automat într-un registru tampon, din care este citit imediat după, astfel încât cei doi octeți provin din același moment de timp. Acest mecanism funcționează însă doar dacă cele două citiri nu sunt separate de o întrerupere care, la rândul ei, accesează un alt registru de 16 biți al aceluiași timer, deoarece registrul tampon este unic și ar fi suprascris.

Observație

Atunci când un registru de 16 biți al lui Timer1 este citit sau scris în programul principal, iar o rutină de întrerupere accesează la rândul ei registre de 16 biți ale aceluiași timer, accesul din programul principal trebuie protejat prin dezactivarea temporară a întreruperilor. Altfel, registrul tampon comun poate fi suprascris între cele două jumătăți ale accesului.

6.3.2 Interogarea indicatorilor de stare

A doua modalitate, numită *interogare* sau *polling*, constă în citirea repetată, într-o buclă, a unui indicator din registrul de stare al timerului, până când acesta capătă valoarea așteptată. Indicatorii uzuali sunt TOVn, care se activează la depășirea număratorului, și OCFnx, care se activează la egalitatea cu registrul de comparare.

Exemplu 6.3 Interogarea unui indicator de stare al timerului

```

1 // asteapta depasirea Timer1 fara a folosi intreruperi
2 while ( !(TIFR1 & (1 << TOV1)) ) {
3     ;                               // bucla goala: se pierde cicluri masina
4 }
5 TIFR1 = (1 << TOV1); // stergerea se face scriind 1 in pozitia
    ↪ indicatorului

```

Interogarea are un avantaj real: precizia obținută este maximă, pentru că programul reacționează la eveniment în câteva cicluri mașină, fără latența introdusă de salvarea contextului la o întrerupere. Are însă și un dezavantaj care o face inutilizabilă în majoritatea aplicațiilor: cât timp bucla se învâрте, procesorul nu poate face nimic altceva. Este exact defectul funcției `delay()`, doar că vizibil mai clar. Interogarea își găsește locul în secvențe scurte și critice, unde nu se poate accepta latența unei întreruperi, sau în programe atât de simple încât nu au altceva de făcut.

Merită observat că indicatorii de stare ai timerelor AVR se șterg printr-o convenție neobișnuită: se scrie valoarea logică 1 în poziția respectivă. O tentativă de a-i șterge prin scrierea unui zero, așa cum ar sugera intuiția, nu are niciun efect, iar programul rămâne blocat la următoarea așteptare.

6.3.3 Folosirea întreruperilor

A treia modalitate, și cea preferată în aplicațiile reale, este ca modulul de timp să anunțe singur procesorul atunci când s-a produs evenimentul așteptat. Programul principal își vede de treabă, iar la momentul potrivit execuția este deviată automat către o rutină de tratare, după care revine exact de unde a fost întreruptă. Costul este o latență de câțiva microsecunde, necesară salvării contextului, dar câștigul este că procesorul rămâne disponibil între evenimente.

Pentru ca o întrerupere de timer să fie efectiv luată în seamă, trebuie îndeplinite trei condiții, iar omiterea oricăreia dintre ele duce la un program care pare corect și totuși nu funcționează. Mai întâi, sursa de întrerupere trebuie validată individual, prin activarea bitului corespunzător din registrul de mască al timerului (TIMSK0, TIMSK1 sau TIMSK2). Apoi, sistemul global de întreruperi trebuie activat, prin instrucțiunea `sei()`, care setează bitul I din registrul de stare al procesorului. În sfârșit, programul trebuie să conțină o rutină de tratare asociată vectorului corect; dacă vectorul rămâne fără rutină, o întrerupere produsă va duce la resetarea microcontrolerului.

Modulele de timp ale ATmega328P pot genera trei categorii de întreruperi:

1. *Overflow Interrupt*, la depășirea numărătorului, adică la trecerea de la valoarea maximă înapoi la zero. Este folosită pentru evenimente periodice a căror perioadă rezultă din rezoluția modulului și din factorul de prescalare.
2. *Compare Match Interrupt*, la egalitatea dintre numărător și un registru de comparație. Perioada devine programabilă, iar în modul CTC ștergerea numărătorului fiind automată, evenimentele sunt riguros echidistante.
3. *Input Capture Interrupt*, la apariția unui front pe pinul de captură. Valoarea numărătorului din acel moment este memorată automat în registrul de captură, ceea ce permite datarea exactă a evenimentelor externe.

O rutină de tratare se scrie folosind macroinstrucțiunea `ISR`, căreia i se dă numele vectorului de întrerupere:

Exemplu 6.4 Structura unei rutine de tratare a întreruperii de timer

```

1 ISR(TIMER1_COMPA_vect) {
2     // cod executat la fiecare egalitate cu OCR1A
3 }
```

Regulile de scriere a unei rutine de întrerupere au fost discutate în capitolul dedicat sistemului de întreruperi, dar merită reamintite aici cele două care se încalcă cel mai des. Rutina trebuie să fie scurtă: tot ce se execută în ea întârzie programul principal și celelalte întreruperi, iar o rutină care durează mai mult decât perioada evenimentului duce la pierderi de evenimente sau la blocarea completă a aplicației. Nu se apelează în ea funcții care depind, la rândul lor, de întreruperi, cum sunt `delay()` sau tipărirea pe interfața serială. A doua

regulă privește variabilele partajate între rutină și programul principal: ele trebuie declarate *volatile*, altfel compilatorul, care nu are cum să știe că valoarea se modifică asincron, poate păstra variabila într-un registru și poate genera o buclă care nu se termină niciodată.

6.4 Utilizarea timerului cu întrerupere

Cel mai frecvent tipar de utilizare a unui timer este generarea unei întreruperi periodice, care devine baza de timp a întregii aplicații. Vom construi acest tipar în doi pași: mai întâi cea mai simplă formă, întreruperea la depășire, apoi forma exactă, cu comparare în modul CTC.

6.4.1 Întreruperea la depășire

Configurarea unui timer pentru întrerupere la depășire cere doar două lucruri: selectarea sursei de tact, care totodată pornește modulul, și validarea sursei de întrerupere. Exemplul următor folosește Timer2, ales pentru că modificarea lui nu afectează contorul de milisecunde.

Exemplu 6.5 Întrerupere la depășirea număratorului Timer2

```

1 void configurareTimer2(void) {
2     TCCR2A = 0x00;                // mod normal: numara pana la 255
    ↪ si revine la 0
3     TCCR2B = (1 << CS20);        // sursa de tact: ceasul
    ↪ sistemului, fara prescalare
4     TCNT2 = 0x00;                // se porneste de la zero
5     TIMSK2 = (1 << TOIE2);      // valideaza intreruperea de
    ↪ depasire
6     sei();                       // valideaza global sistemul de
    ↪ intreruperi
7 }
8
9 ISR(TIMER2_OVF_vect) {
10    PORTB ^= (1 << PB5);         // comuta starea pinului, pentru
    ↪ vizualizare
11 }
```

Ceea ce se obține în acest fel merită analizat cantitativ, pentru că rezultatul este de obicei o surpriză. Numărătorul are 8 biți, deci parcurge 256 de stări, iar ceasul este de 16 MHz, deci o stare durează 62,5 ns. Depășirea se produce la fiecare $256 \times 62,5 \text{ ns} = 16 \mu\text{s}$, adică de 62 500 de ori pe secundă. Pinul comandat, care își schimbă starea la fiecare întrerupere, va oscila la jumătate din această frecvență, adică la 31,25 kHz. Nu este ceea ce ar dori cineva care încearcă să clipească un LED. Mai grav, la fiecare 16 μs procesorul este obligat să salveze contextul, să execute rutina și să îl restaureze, ceea ce înseamnă că o fracțiune importantă din puterea de calcul este consumată de întreruperi. Concluzia este că un numărator de 8 biți alimentat direct cu ceasul de sistem este practic inutilizabil pentru temporizări, iar

soluția este circuitul de prescalare.

6.5 Circuitul de prescalare

Prescalerul este un divizor de frecvență intercalat între sursa de tact și intrarea numărătorului. El este realizat ca un lanț de bistabile care împart succesiv frecvența la doi, iar un multiplexor comandat de biții de configurare alege de la care etaj al lanțului se preia semnalul. De aici rezultă și forma valorilor disponibile: ele sunt de forma 2^P , unde P este numărul de etaje folosite, ceea ce explică de ce factorii de prescalare nu pot fi aleși arbitrar, ci doar din lista fixată de constructor. Pentru Timer0 și Timer1 ai lui ATmega328P această listă este 1, 8, 64, 256 și 1024, iar pentru Timer2, care dispune de un lanț mai lung de derivații, ea cuprinde în plus valorile 32 și 128. Lista exactă se verifică întotdeauna în foaia de catalog, pentru că diferă de la un modul la altul chiar în cadrul aceluiași microcontroler.

Efectul prescalerului este dublu. Pe de o parte, el prelungește durata până la depășire, făcând posibile temporizări lungi cu un numărător mic. Pe de altă parte, el înrăutățește rezoluția: dacă un impuls de tact al numărătorului durează 64 de perioade de ceas, atunci momentele de timp nu mai pot fi distinse decât cu o precizie de 64 de perioade, adică de $4 \mu\text{s}$ la 16 MHz. Există așadar un compromis, iar regula practică este să se aleagă cel mai mic factor de prescalare care mai permite acoperirea intervalului dorit, pentru a păstra rezoluția cea mai bună.

Figura 6.2 arată lanțul complet, de la oscilator până la comparator și la ieșire.

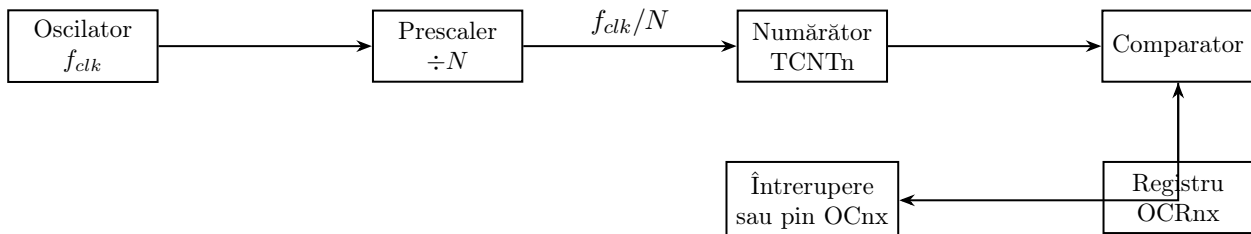


Figura 6.2 Lanțul de prelucrare a semnalului de tact într-un modul de tip timer: ceasul este divizat de prescaler, incrementează numărătorul, iar comparatorul compară continuu valoarea acestuia cu registrul de comparare.

6.5.1 Legătura dintre frecvență, prescalare și perioadă

Fie f_{clk} frecvența ceasului aplicat modulului, N factorul de prescalare și T_{tick} durata unui impuls al numărătorului. Atunci

$$T_{tick} = \frac{N}{f_{clk}}.$$

Dacă modulul lucrează în mod normal, cu rezoluția de n biți, o depășire se produce după 2^n impulsuri, deci perioada evenimentelor este

$$T_{ovf} = \frac{N \cdot 2^n}{f_{clk}}.$$

Dacă modulul lucrează în mod CTC, numărătorul se șterge la egalitatea cu registrul de comparare, deci parcurge $OCRnA + 1$ impulsuri pe perioadă - valorile de la 0 la $OCRnA$ inclusiv - iar perioada devine

$$T = \frac{N \cdot (OCRnA + 1)}{f_{clk}}, \quad f = \frac{f_{clk}}{N \cdot (OCRnA + 1)}.$$

Relația se inversează imediat, pentru a afla valoarea de comparare necesară unei perioade impuse:

$$OCRnA = \frac{f_{clk} \cdot T}{N} - 1 = \frac{f_{clk}}{N \cdot f} - 1.$$

Această formulă este instrumentul de bază al întregului capitol. Rezultatul ei trebuie să fie un număr întreg cuprins în domeniul registrului de comparare, adică cel mult 255 pentru un timer de 8 biți și cel mult 65535 pentru unul de 16 biți. Dacă rezultatul depășește domeniul, se alege un factor de prescalare mai mare; dacă rezultatul este foarte mic, de câteva unități, se alege unul mai mic, pentru că altfel eroarea de cuantizare devine importantă. Dacă rezultatul nu este întreg, frecvența cerută nu poate fi obținută exact, iar eroarea relativă se estimează comparând perioada obținută cu valoarea rotunjită cu perioada dorită.

Exemplu

Să se configureze un timer al microcontrolerului ATmega328P, alimentat de la un oscilator de 16 MHz, astfel încât să genereze o întrerupere la fiecare milisecundă.

Rezolvare. Perioada cerută este $T = 1$ ms, deci frecvența evenimentelor este $f = 1000$ Hz. Numărul total de impulsuri de ceas care încap într-o milisecundă este

$$\frac{f_{clk}}{f} = \frac{16 \cdot 10^6}{1000} = 16\,000.$$

Aceste 16 000 de impulsuri trebuie împărțite între prescaler și numărător.

Se încearcă mai întâi varianta fără prescalare, $N = 1$. Rezultă $OCRnA = 16\,000 - 1 = 15\,999$, valoare care nu încapă într-un registru de 8 biți, dar încapă confortabil în cei 16 biți ai lui Timer1. Soluția este deci posibilă pe Timer1 și oferă rezoluția maximă, un impuls la fiecare 62,5 ns.

Dacă se dorește însă folosirea unui timer de 8 biți, valoarea de comparare nu poate depăși 255, deci este nevoie de prescalare. Se caută cel mai mic N pentru care $16\,000/N - 1 \leq 255$, adică $N \geq 62,5$. Din lista valorilor disponibile, prima care satisface condiția este $N = 64$. Cu ea se obține

$$OCRnA = \frac{16 \cdot 10^6}{64 \cdot 1000} - 1 = 250 - 1 = 249.$$

Rezultatul este întreg și încapă în 8 biți, deci perioada de o milisecundă se obține exact, fără eroare. Un impuls al numărătorului durează $64/16 \cdot 10^6 = 4 \mu s$, iar cele 250 de

impulsuri numărate dau exact $1000 \mu\text{s}$.
 Verificare: $250 \times 4 \mu\text{s} = 1000 \mu\text{s} = 1 \text{ ms}$.

Codul care pune în practică acest calcul, folosind Timer1 cu prescalare 64 și valoarea de comparare 249, este următorul.

Exemplu 6.6 Întrerupere periodică exactă la 1 ms, în mod CTC

```

1  volatile uint32_t milisecunde = 0;
2
3  void configurareBazaDeTimp(void) {
4      cli(); // opreste
           ↪ intreruperile pe durata configurarii
5      TCCR1A = 0x00; // fara actiune pe pini
6      TCCR1B = (1 << WGM12); // mod CTC, TOP dat de
           ↪ OCR1A
7      OCR1A = 249; // 16 MHz / (64 * 250)
           ↪ = 1000 Hz
8      TCCR1B |= (1 << CS11) | (1 << CS10); // prescaler 64:
           ↪ porneste numaratorul
9      TIMSK1 = (1 << OCIE1A); // valideaza
           ↪ intreruperea de comparare A
10     TCNT1 = 0;
11     sei();
12 }
13
14 ISR(TIMER1_COMPA_vect) {
15     milisecunde++; // baza de timp proprie
           ↪ aplicatiei
16 }
```

Ordinea operațiilor din această funcție nu este întâmplătoare. Biții de selecție a sursei de tact se scriu ultimii, pentru că scrierea lor este cea care porneste efectiv numărătorul; dacă modulul ar fi pornit înainte de a fi complet configurat, prima perioadă ar avea o durată greșită. Din același motiv întreruperile globale sunt oprite pe durata configurării, iar numărătorul este șters explicit înainte de pornire.

Comparând acest rezultat cu întreruperea la depășire fără prescalare din secțiunea anterioară, câștigul este evident. Frecvența întreruperilor a scăzut de la 62 500 la 1000 pe secundă, deci sarcina impusă procesorului s-a redus de peste șaizeci de ori, iar perioada este exact cea dorită, nu una impusă de rezoluția numărătorului.

6.6 Caracteristicile semnalelor periodice

Înainte de a trece la generarea semnalelor PWM, este necesar să fixăm câteva noțiuni despre semnalele periodice, pentru că ele constituie limbajul în care se descriu atât intrările, cât și ieșirile modulelor de timp.

Un semnal $x(t)$ se numește *periodic*, de *perioadă* T , dacă valoarea sa se repetă identic după fiecare interval T , adică

$$x(t) = x(t + T), \quad \forall t.$$

Cea mai mică valoare pozitivă a lui T care satisface relația se numește *perioadă fundamentală* și este cea la care ne referim în mod obișnuit. *Frecvența* este numărul de repetări ale ciclului într-o unitate de timp și se măsoară în hertzi, adică în cicluri pe secundă. Cele două mărimi sunt inverse una alteia:

$$f = \frac{1}{T}, \quad T = \frac{1}{f}.$$

Un semnal de 1 Hz are perioada de o secundă, unul de 1 kHz are perioada de o milisecundă, iar unul de 1 MHz are perioada de o microsecundă. Această corespondență trebuie să devină un reflex, pentru că toate calculele de configurare a timerelor trec prin ea.

Pentru un semnal dreptunghiular, cele două mărimi de mai sus nu sunt suficiente, pentru că nu spun nimic despre proporția dintre durata stării active și cea a stării inactive. Această proporție se numește *factor de umplere* (în engleză *duty cycle*) și se definește ca raportul dintre durata stării active T_{on} și perioada totală:

$$FU = \frac{T_{on}}{T} \cdot 100 [\%].$$

Un factor de umplere de 0 % înseamnă un semnal permanent inactiv, unul de 100 % un semnal permanent activ, iar 50 % corespunde unui semnal simetric, cu durate egale ale celor două stări. La aceste trei mărimi se adaugă *amplitudinea*, adică tensiunea corespunzătoare stării active; la un microcontroler ea este impusă de tensiunea de alimentare și nu poate fi modificată prin software.

6.6.1 De ce factorul de umplere este mărimea care contează

Ideea centrală, care explică de ce PWM este atât de răspândit, este următoarea: dacă frecvența semnalului este suficient de mare în raport cu inerția sarcinii comandate, sarcina nu mai percepe comutările individuale, ci numai valoarea medie a semnalului, iar aceasta este proporțională cu factorul de umplere. Pentru un semnal dreptunghiular care comută între 0 și V_{cc} ,

$$V_{med} = V_{cc} \cdot \frac{T_{on}}{T} = V_{cc} \cdot \frac{FU}{100}.$$

La un motor de curent continuu, inerția este de natură mecanică și electrică: rotorul are

masă, deci nu își poate schimba turația în câteva zeci de microsecunde, iar înfășurarea are inductanță, deci curentul prin ea nu poate sări instantaneu. Dacă motorul este alimentat cu impulsuri la câteva kilohertzi, curentul prin înfășurare se netezește, iar cuplul dezvoltat corespunde valorii medii a tensiunii aplicate. Rezultatul este că turația se reglează continuu, de la zero la maxim, doar prin modificarea factorului de umplere.

La un LED, inerția este de natură fiziologică: ochiul integrează lumina pe un interval de ordinul a douăzeci de milisecunde, astfel încât un LED comandat cu impulsuri de peste o sută de hertzi este perceput ca luminând continuu, cu o strălucire proporțională cu factorul de umplere. Sub această frecvență apare pâlpâirea, supărătoare și obositoare, iar acesta este motivul pentru care frecvențele PWM implicite ale plăcilor Arduino, de aproximativ 490 Hz și 980 Hz, sunt alese peste pragul de percepție.

Ceea ce face metoda atât de atractivă este randamentul. Aceeași reducere a puterii s-ar putea obține punând un rezistor în serie sau folosind un tranzistor în regim liniar, dar în ambele cazuri diferența de putere s-ar transforma în căldură. Comutatorul comandat în PWM este fie complet deschis, caz în care nu îl străbate curent, fie complet închis, caz în care căderea de tensiune pe el este aproape nulă; în ambele situații puterea disipată pe comutator este mică. De aceea comanda în PWM permite reglarea puterii pe sarcini de zeci sau sute de wați cu un tranzistor care abia se încălzește.

Observație

Modularea în durată nu schimbă amplitudinea semnalului, ci numai valoarea lui medie. Sarcina trebuie să aibă suficientă inerție - mecanică, termică, optică sau introdusă printr-un filtru - pentru a media semnalul. Aplicat unei sarcini rapide, fără filtrare, un semnal PWM rămâne ceea ce este: o succesiune de impulsuri complete.

6.7 Generarea unui semnal cu factor de umplere variabil (PWM)

Un modul de timp configurat în regim PWM produce, fără nicio intervenție a procesorului, un semnal dreptunghiular de frecvență fixă și factor de umplere programabil. Parametrii semnalului sunt cei definiți în secțiunea anterioară: perioada T , dată de frecvența ceasului, de factorul de prescalare și de valoarea TOP; frecvența $f = 1/T$; amplitudinea, impusă de tensiunea de alimentare; și factorul de umplere, dat de valoarea înscrisă în registrul de comparare.

Figura 6.3 arată același semnal la două valori ale factorului de umplere, împreună cu valoarea medie corespunzătoare. Se observă că perioada rămâne neschimbată, ceea ce se modifică fiind doar lățimea impulsului, iar odată cu ea și nivelul mediu.

Frecvența semnalului PWM se poate modifica în trei feluri, cu consecințe diferite. Cea mai comodă este schimbarea factorului de prescalare, care împarte frecvența la valorile discrete disponibile; metoda este simplă, dar nu permite reglaje fine, pentru că valorile sunt puteri ale lui doi. A doua este schimbarea valorii TOP, posibilă în modurile în care TOP este dat de un registru; metoda permite reglarea aproape continuă a frecvenței, dar reduce rezoluția factorului de umplere, pentru că numărul de trepte disponibile este chiar TOP. A

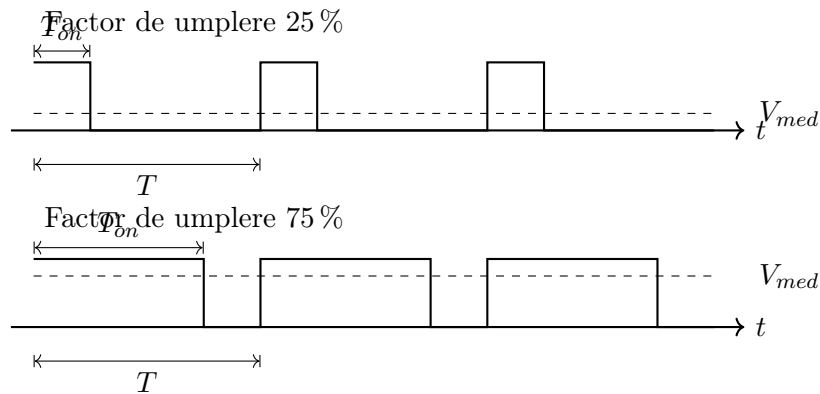


Figura 6.3 Semnal PWM la două valori ale factorului de umplere. Perioada rămâne constantă, se modifică numai lățimea impulsului și, odată cu ea, valoarea medie.

treia este modificarea frecvenței ceasului de sistem, care afectează însă tot ce depinde de timp în microcontroler, de la vitezele de comunicație până la funcțiile de bibliotecă, și de aceea se folosește doar în situații bine justificate, de regulă pentru reducerea consumului.

6.8 Implementarea PWM

6.8.1 PWM rapid

Mecanismul de generare este remarcabil de simplu, ceea ce explică de ce este implementat în hardware pe aproape orice microcontroler. Numărătorul crește continuu de la zero la TOP. Un comparator compară permanent valoarea lui cu conținutul registrului de comparare. Un bloc logic comandă pinul de ieșire astfel: la revenirea numărătorului la zero, pinul este trecut în starea activă; la egalitatea dintre numărător și registrul de comparare, pinul este trecut în starea inactivă. Rezultă un impuls a cărui lățime este proporțională cu valoarea înscrisă în registrul de comparare.

Perioada semnalului este dată de numărul de impulsuri de tact necesare numărătorului pentru a parcurge domeniul complet, adică $TOP + 1$ stări. Frecvența rezultată este

$$f_{PWM} = \frac{f_{clk}}{N \cdot (TOP + 1)},$$

iar factorul de umplere este

$$FU = \frac{OCR_{nx} + 1}{TOP + 1} \cdot 100 [\%].$$

Pentru configurația cea mai des întâlnită, cu TOP fixat la 255 și rezoluția de 8 biți, frecvența devine $f_{clk}/(N \cdot 256)$. Cu ceasul de 16 MHz și prescalarea 64, rezultă $16 \cdot 10^6 / (64 \cdot 256) \approx 976,6$ Hz, adică exact frecvența pe care o are semnalul produs de `analogWrite()` pe pinii 5 și 6 ai plăcii Arduino Uno.

Rezoluția factorului de umplere, exprimată în biți, se obține din numărul de trepte distincte disponibile:

$$R = \log_2(\text{TOP} + 1).$$

Cu $\text{TOP} = 255$ avem 256 de trepte, deci 8 biți, adică un pas de aproximativ 0,39 % din perioadă. Aceasta este rezoluția pe care o oferă `analogWrite()`, ale cărei argumente sunt numere de la 0 la 255. Dacă aplicația cere mai mult - de exemplu comanda fină a unui servomotor - se folosește Timer1 cu TOP dat de registrul ICR1, care poate ajunge la 65535, deci până la 16 biți de rezoluție.

Relația dintre frecvență și rezoluție este de compromis și rezultă imediat din formule: produsul dintre frecvența PWM și numărul de trepte este constant, egal cu f_{clk}/N . Nu se poate obține simultan o frecvență mare și o rezoluție mare decât mărinnd frecvența ceasului sau reducând prescalarea.

6.8.2 PWM cu fază corectă

În acest mod, numărătorul urcă de la zero la TOP și coboară înapoi la zero, descriind o rampă triunghiulară. Ieșirea comută la fiecare trecere prin valoarea de comparare, o dată pe fiecare rampă, ceea ce dă un impuls centrat pe același moment al perioadei indiferent de lățimea sa. Deoarece numărătorul parcurge domeniul de două ori pe perioadă, durata unui ciclu este

$$T_{PWM} = 2 \cdot \text{TOP} \cdot T_{tick} = \frac{2 \cdot \text{TOP} \cdot N}{f_{clk}}, \quad f_{PWM} = \frac{1}{T_{PWM}} = \frac{f_{clk}}{2 \cdot N \cdot \text{TOP}}.$$

În implementarea concretă a modulelor AVR de 8 biți, numărul de impulsuri pe perioadă este 510, adică $2 \cdot 255$, pentru că valorile extreme nu sunt numărate de două ori. Cu ceasul de 16 MHz și prescalarea 64 rezultă $16 \cdot 10^6 / (64 \cdot 510) \approx 490,2$ Hz, adică frecvența pe care o are semnalul produs de `analogWrite()` pe pinii 3, 9, 10 și 11. Se observă că, la aceeași rezoluție și același prescaler, PWM-ul cu fază corectă are frecvența aproximativ înjumătățită față de PWM-ul rapid - acesta este prețul plătit pentru fixarea fazei.

Alegerea între cele două moduri urmează o regulă practică. PWM-ul rapid este potrivit pentru marea majoritate a aplicațiilor: iluminare, ventilatoare, motoare simple de curent continuu, generarea unei tensiuni medii filtrate. PWM-ul cu fază corectă se preferă acolo unde deplasarea centrului impulsului ar produce perturbații: comanda motoarelor fără perii, punțile în H cu comandă complementară, aplicațiile audio în care asimetria s-ar traduce în distorsiune.

6.8.3 Configurarea directă a registrelor

Exemplul următor configurează Timer2 în mod PWM rapid, cu TOP fixat la 255 și ieșire neinvertată pe pinul OC2A, care pe placa Arduino Uno corespunde pinului digital 11.

Exemplu 6.7 PWM rapid pe Timer2, cu factor de umplere reglabil

```
1 void configurarePWM2(void) {
```

```

2   DDRB |= (1 << PB3);                                // OC2A este pinul
    ↪ PB3 (pinul 11 pe Uno)
3
4   // WGM21:WGM20 = 11 → Fast PWM cu TOP = 0xFF
5   // COM2A1:COM2A0 = 10 → ieșire neinvertată: HIGH la 0, LOW la
    ↪ egalitate
6   TCCR2A = (1 << WGM21) | (1 << WGM20) | (1 << COM2A1);
7
8   // CS22:CS21:CS20 = 100 → prescaler 64, deci 16 MHz / (64 *
    ↪ 256) = 976,6 Hz
9   TCCR2B = (1 << CS22);
10
11  OCR2A = 25;                                          // 26 / 256 =
    ↪ aproximativ 10 % umplere
12 }
13
14 void setFactorUmplere(uint8_t procent) {
15     // conversie procent → valoare de comparare pe 8 biți
16     OCR2A = (uint16_t)procent * 255 / 100;
17 }

```

Cei doi biți COM2A1 și COM2A0 determină ce face blocul logic de ieșire la egalitate. Combinația folosită mai sus produce un semnal neinvertat: pinul este activ de la începutul perioadei până la egalitate. Combinația complementară produce un semnal inversat, util atunci când sarcina este comandată prin nivel jos sau când se dorește o pereche de semnale în opoziție. Dacă ambii biți sunt zero, blocul de ieșire este deconectat, iar pinul rămâne un port de uz general - situație în care modulul continuă totuși să numere și poate genera întreruperi.

6.8.4 Legătura cu funcția analogWrite()

Funcția `analogWrite(pin, valoare)` din biblioteca Arduino nu face altceva decât ceea ce s-a descris mai sus. Ea identifică timerul asociat pinului cerut, activează blocul de comparare corespunzător prin scrierea biților COM și înscrie argumentul în registrul de comparare. Numele funcției este, de altfel, nefericit ales: ieșirea nu este analogică, ci tot digitală, comutând între 0 V și 5 V; doar valoarea ei medie variază continuu.

De aici decurg câteva consecințe utile în practică. Argumentul are opt biți pentru că registrele de comparare ale timerelor de 8 biți au opt biți; nu este o limitare a funcției, ci a hardware-ului pe care se sprijină. Frecvența nu poate fi schimbată prin `analogWrite()`, pentru că funcția nu atinge biții de prescalare; ea se schimbă doar scriind direct în registrele de control, cu grija discutată anterior privind Timer0. Cazurile particulare `analogWrite(pin, 0)` și `analogWrite(pin, 255)` sunt tratate special în bibliotecă, prin deconectarea blocului

de comparare și comanda directă a pinului, pentru a garanta un nivel constant. În sfârșit, numai șase pini ai plăcii Uno pot fi folosiți cu această funcție, pentru că numai șase ieșiri de comparare există în hardware - câte două pentru fiecare dintre cele trei timere.

Exemplu 6.8 Reglarea intensitatii unui LED prin analogWrite

```

1  const uint8_t LED = 9;           // pin comandat de Timer1
2
3  void setup() {
4      pinMode(LED, OUTPUT);
5  }
6
7  void loop() {
8      for (int v = 0; v <= 255; v++) {
9          analogWrite(LED, v);      // crește treptat factorul de
          ↪ umplere
10         delay(5);
11     }
12     for (int v = 255; v >= 0; v--) {
13         analogWrite(LED, v);
14         delay(5);
15     }
16 }
```

Un ultim amănunt de implementare merită menționat. Registrele TOP și de comparare sunt scrise de program în timp ce numărătorul funcționează. Dacă valoarea de comparare ar fi modificată exact atunci când numărătorul tocmai a depășit-o, impulsul curent ar fi deformat, iar în cazuri nefavorabile ar lipsi cu totul sau s-ar prelungi peste o perioadă întreagă. Pentru a evita acest lucru, modulele AVR folosesc registre-tampon: valoarea scrisă de program este reținută și transferată în registrul activ numai la începutul unei noi perioade. Mecanismul elimină cele mai multe erori de sincronizare, dar nu pe toate; în modurile în care TOP este variabil, o modificare făcută la un moment nepotrivit poate totuși produce un ciclu de durată greșită.

6.9 Capturarea evenimentelor externe

Toate mecanismele discutate până acum privesc generarea de evenimente la momente alese de programator. Problema inversă - determinarea momentului exact la care s-a produs un eveniment extern - este cel puțin la fel de importantă și este rezolvată de unitatea de *captură a intrării* (*input capture*).

6.9.1 Mecanismul capturii

Ideea este simplă și extrem de eficientă. Un pin dedicat, notat ICP1 la ATmega328P și corespunzând pinului digital 8 al plăcii Uno, este supravegheat permanent de hardware. La

apariția unui front de sensul configurat, valoarea instantanee a numărătorului este copiată automat, în hardware, în registrul de captură ICR1; totodată se activează indicatorul ICF1 și, dacă întreruperea corespunzătoare a fost validată, se generează o cerere de întrerupere.

Esențial este că memorarea valorii se face în momentul frontului, nu în momentul în care programul ajunge să reacționeze. Aceasta este deosebirea de fond față de o întrerupere externă obișnuită, în care rutina de tratare ar trebui să citească ea însăși numărătorul; într-o astfel de soluție, valoarea citită ar fi afectată de latența întreruperii, adică de timpul de terminare a instrucțiunii curente, de salvarea contextului și de eventuala așteptare după o altă rutină aflată în execuție. Această latență variază de la un caz la altul, cu zeci de cicluri, și nu poate fi compensată prin calcul. La captura hardware, ea nu mai influențează rezultatul: întreruperea poate întârzia oricât, atât timp cât rutina apucă să citească ICR1 înainte de următorul front, valoarea memorată rămâne exactă la nivelul unui impuls de tact. Rutina de tratare nu mai are rolul de a măsura, ci doar pe acela de a ridica un rezultat deja măsurat.

Precizia obținută este așadar de un impuls al numărătorului, adică N/f_{clk} . Mai exact, momentul real al evenimentului și momentul înregistrat diferă cu mai puțin de o perioadă a tactului numărătorului, la care se adaugă întârzierea introdusă de circuitul de sincronizare a intrării, care preia semnalul asincron de pe pin și îl aduce în domeniul de ceas al modulului. Dacă notăm cu P factorul de prescalare și cu d_{in} întârzierea de sincronizare exprimată în perioade de ceas, eroarea comisă la datarea unui eveniment se încadrează în intervalul

$$t_{eveniment} - t_{capt} \in (-d_{in}^{max}, P - 1 - d_{in}^{min}),$$

iar eroarea maximă la măsurarea intervalului dintre două evenimente succesive, care presupune două capturi, este

$$\varepsilon_{max} = d_{in}^{max} + P - 1 - d_{in}^{min}.$$

Se observă că eroarea crește direct cu factorul de prescalare, ceea ce confirmă regula enunțată anterior: pentru capturi se alege cel mai mic prescaler posibil. Limita în cealaltă direcție vine din domeniul numărătorului. Intervalul maxim care poate fi măsurat fără ambiguitate, adică până la o depășire, este

$$\Delta t_{max} = (2^r - 1) \cdot \frac{P}{f_{clk}},$$

unde r este rezoluția timerului. Alegerea prescalerului este deci un compromis între precizie și domeniu: prea mic, și numărătorul se umple înainte de al doilea front; prea mare, și rezoluția devine grosolană.

Figura 6.4 ilustrează mecanismul pentru măsurarea lățimii unui impuls: la frontul crescător se memorează o primă valoare, la cel descrescător a doua, iar diferența lor, înmulțită cu durata unui impuls de tact, dă lățimea căutată.

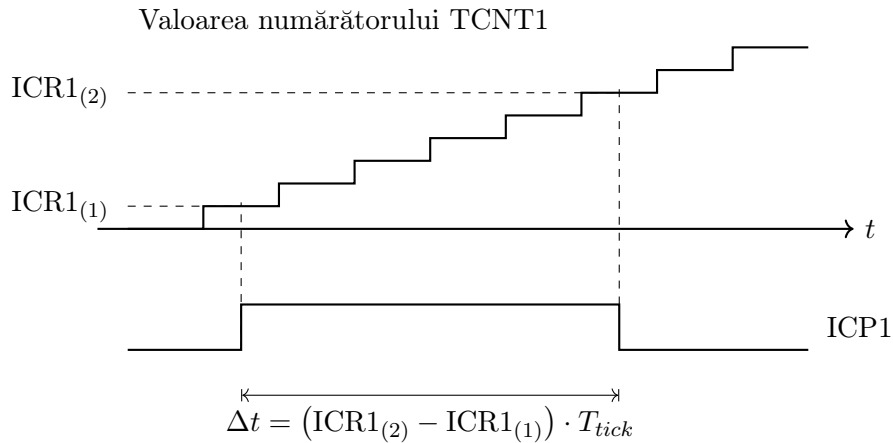


Figura 6.4 Cronograma unei capturi: la fiecare front al semnalului aplicat pe pinul ICP1, valoarea instantanee a număratorului este memorată în registrul ICR1.

6.9.2 Configurarea unei măsurători

Realizarea unei măsurători prin captură urmează, indiferent de aplicație, aceiași pași. Se alege un timer care are unitate de captură - la ATmega328P doar Timer1 - și un factor de prescalare potrivit cu ordinul de mărime al duratelor măsurate. Se configurează sensul frontului activ, prin bitul ICES1 din TCCR1B: valoarea 1 selectează frontul crescător, valoarea 0 pe cel descrescător. Se validează întreruperea de captură prin bitul ICIE1 din TIMSK1 și se scrie rutina asociată vectorului TIMER1_CAPT_vect. În rutină, prima valoare capturată se memorează, se inversează sensul frontului activ, iar la a doua captură se face diferența.

Modulul dispune și de un filtru de zgomot, activat prin bitul ICNC1. Când filtrul este activ, hardware-ul eșantionează pinul de patru ori consecutiv și acceptă frontul numai dacă toate cele patru eșantioane sunt identice, ceea ce elimină impulsurile parazite scurte cu prețul unei întârzieri fixe de patru perioade de ceas. Fiind o întârziere constantă, ea nu afectează măsurarea unei diferențe de timp, pentru că se anulează prin scădere.

6.9.3 Aplicație: senzorul cu ultrasunete

Un exemplu care arată limpede utilitatea capturii este măsurarea distanței cu un senzor cu ultrasunete de tip HC-SR04. Principiul de funcționare al senzorului este următorul: microcontrolerul aplică pe intrarea de declanșare un impuls scurt, senzorul emite un tren de ultrasunete și apoi activează linia de ieșire; linia rămâne activă exact atât timp cât durează călătoria dus-întors a unei sonore până la obstacol. Distanța se obține din durata măsurată, cunoscând viteza sunetului în aer, aproximativ 343 m/s la temperatura camerei, și ținând seama că unda parcurge de două ori distanța:

$$d = \frac{v_{\text{sunet}} \cdot \Delta t}{2}.$$

Toată problema se reduce astfel la măsurarea cât mai exactă a lățimii unui impuls, iar aceasta este exact ceea ce face unitatea de captură. Soluția uzuală în programele de

început folosește funcția `pulseIn()`, care măsoară lățimea impulsului numărând într-o buclă software. Ea funcționează, dar are două defecte grave: blochează procesorul pe toată durata măsurătorii, care poate ajunge la zeci de milisecunde dacă nu există obstacol, și dă rezultate afectate de orice întrerupere survenită în acel interval. Cu unitatea de captură, măsurătoarea se face în hardware, iar programul nu pierde decât cele câteva microsecunde ale celor două rutine de întrerupere.

Exemplu 6.9 Masurarea latimii unui impuls prin captura pe Timer1

```

1  volatile uint16_t captura1 = 0;
2  volatile uint16_t latime    = 0;
3  volatile uint8_t  gata      = 0;
4
5  void configurareCaptura(void) {
6      TCCR1A = 0x00;                                     // mod normal, fara
        ↪ actiune pe pini
7      // ICNC1: filtru de zgomot activ; ICES1: front crescator; CS11:
        ↪ prescaler 8
8      TCCR1B = (1 << ICNC1) | (1 << ICES1) | (1 << CS11);
9      TIFR1  = (1 << ICF1);                             // sterge un eventual
        ↪ indicator vechi
10     TIMSK1 = (1 << ICIE1);                             // valideaza
        ↪ intreruperea de captura
11     sei();
12 }
13
14 ISR(TIMER1_CAPT_vect) {
15     if (TCCR1B & (1 << ICES1)) {                       // a fost front
        ↪ crescator
16         captura1 = ICR1;                                // se retine
        ↪ inceputul impulsului
17         TCCR1B &= ~(1 << ICES1);                       // urmatorul front:
        ↪ descrescator
18     } else {                                             // a fost front
        ↪ descrescator
19         latime = ICR1 - captura1;                       // diferenta, in
        ↪ impulsuri de tact
20         TCCR1B |= (1 << ICES1);                         // revenire la front
        ↪ crescator
21         gata = 1;
22     }
23     TIFR1 = (1 << ICF1);                               // sterge indicatorul

```

24 } ↪ dupa comutare

Câteva detalii din acest cod merită comentate. Scăderea `ICR1 - captura1` se face pe aritmetică de 16 biți fără semn, ceea ce înseamnă că rezultatul rămâne corect chiar dacă numărătorul a trecut prin zero între cele două capturi, atât timp cât nu s-a produs mai mult de o depășire. Indicatorul `ICF1` este șters explicit după schimbarea sensului frontului, pentru că operația de comutare poate ea însăși activa indicatorul, ceea ce ar produce o întrerupere falsă imediat după ieșirea din rutină. Variabilele partajate sunt declarate *volatile*, iar citirea lor în programul principal trebuie făcută cu întreruperile oprite temporar, pentru că o variabilă de 16 biți se citește în doi pași și ar putea fi modificată între ei.

Exemplu

Cu ce rezoluție și pe ce domeniu se poate măsura distanța, folosind `Timer1` la 16 MHz cu factorul de prescalare 8, în montajul descris mai sus?

Rezolvare. Durata unui impuls al numărătorului este

$$T_{tick} = \frac{N}{f_{clk}} = \frac{8}{16 \cdot 10^6} = 0,5 \mu s.$$

Unui singur impuls îi corespunde o variație de distanță

$$\Delta d = \frac{v_{sunet} \cdot T_{tick}}{2} = \frac{343 \cdot 0,5 \cdot 10^{-6}}{2} \approx 8,6 \cdot 10^{-5} \text{ m},$$

adică aproximativ 0,086 mm, cu mult sub precizia proprie a senzorului. Rezoluția măsurării timpului nu este, așadar, factorul limitativ.

Domeniul maxim rezultă din capacitatea numărătorului de 16 biți:

$$\Delta t_{max} = 65535 \cdot 0,5 \mu s \approx 32,8 \text{ ms},$$

căreia îi corespunde distanța

$$d_{max} = \frac{343 \cdot 32,8 \cdot 10^{-3}}{2} \approx 5,6 \text{ m},$$

mult peste domeniul util al senzorului, care este de câțiva metri. Alegerea prescalerului 8 este deci potrivită: asigură o rezoluție inutil de bună și un domeniu suficient. Un prescaler de 1 ar da o rezoluție dublă, dar ar reduce domeniul la 4,1 ms, adică la circa 70 cm, ceea ce ar fi insuficient.

6.9.4 Alte utilizări ale capturii

Aceeași unitate rezolvă o întreagă familie de probleme de măsurare. Măsurarea frecvenței unui semnal se face capturând două fronturi de același sens și inversând intervalul dintre ele; metoda este superioară numărării impulsurilor într-o fereastră fixă atunci când frecvența

este joasă, pentru că nu cere o fereastră lungă. Decodarea semnalelor de la o telecomandă în infraroșu, la care informația este codificată chiar în lățimea impulsurilor, se face natural prin capturi succesive. Măsurarea turației unui arbore prevăzut cu un traductor de proximitate, determinarea fazei dintre două semnale sau recepția semnalului de comandă al unui servomotor sunt toate variante ale aceluiași tipar.

Limitarea principală a ATmega328P este că dispune de o singură unitate de captură, asociată lui Timer1. Dacă aplicația trebuie să măsoare simultan mai multe semnale, soluțiile sunt fie multiplexarea în timp, dacă semnalele nu se suprapun, fie folosirea întreruperilor externe cu citirea numărătorului în rutină, acceptând eroarea introdusă de latență, fie trecerea la un microcontroler cu mai multe unități de captură.

6.10 Concluzii

Modulele de tip timer transformă timpul dintr-un efect secundar al execuției programului într-o mărime pe care microcontrolerul o poate măsura și impune cu precizia oscilatorului său. Toate funcțiile lor derivă dintr-un singur mecanism elementar - un numărător care avansează în hardware, alimentat printr-un prescaler dintr-o sursă de tact stabilă - la care se adaugă un comparator și un bloc de captură.

Din acest mecanism decurg cele trei familii de utilizări discutate. Comparând numărătorul cu un prag și ștergându-l automat, se obțin întreruperi periodice exacte, care înlocuiesc așteptarea inutilă din `delay()` cu o bază de timp pe care programul o poate consulta ori-când. Comandând un pin de ieșire în funcție de aceeași comparație, se obține un semnal cu factor de umplere reglabil, care permite reglarea puterii medii pe o sarcină cu randament ridicat. Memorând valoarea numărătorului la apariția unui front extern, se obține datarea exactă a evenimentelor, cu o precizie pe care software-ul nu o poate atinge.

Ce trebuie reținut, dincolo de nume de registre, este disciplina calculului: frecvența de tact, factorul de prescalare și valoarea de comparare determină împreună perioada obținută, iar alegerea lor este un compromis între rezoluție și domeniu. Și, la fel de important, faptul că modulele de timp sunt resurse partajate cu bibliotecile: orice reconfigurare a lor trebuie făcută știind ce funcții încetează, în acel moment, să mai spună adevărul.

Întrebări de verificare

1. Explicați deosebirea dintre regimul de timer și cel de counter al aceluiași modul hardware. Ce se modifică în configurare atunci când se trece de la unul la celălalt și ce semnificație are, în fiecare caz, valoarea citită din registrul de contor?
2. De ce se spune că funcția `delay()` este o risipă de resurse? Descrieți concret ce se pierde în timpul așteptării și arătați cum se poate rescrie, folosind `millis()`, un program care trebuie să clipească un LED la fiecare 500 ms și să citească un senzor la fiecare 200 ms.
3. Un numărător de 8 biți este alimentat direct cu ceasul de 16 MHz al microcontrolerului. După cât timp se produce depășirea și de câte ori pe secundă ar fi apelată rutina de întrerupere asociată? Comentați dacă o asemenea configurație este utilizabilă în

practică.

4. Explicați cum este construit un prescaler și de ce factorii de prescalare disponibili sunt puteri ale lui doi și nu valori arbitrare. Ce se pierde atunci când se alege un factor de prescalare mai mare decât este necesar?
5. Se cere o întrerupere la fiecare 10 ms, folosind Timer1 al unui ATmega328P la 16 MHz în mod CTC. Calculați, pentru fiecare factor de prescalare disponibil, valoarea necesară a registrului `OCR1A` și indicați care variante sunt realizabile și care nu, justificând.
6. De ce modificarea factorului de prescalare al lui Timer0 pe o placă Arduino Uno afectează funcționarea programului chiar dacă acesta nu apelează niciodată `delay()`? Ce funcții devin incorecte și de ce eroarea este greu de depistat?
7. Comparați modul normal cu modul CTC din punctul de vedere al preciziei intervalelor generate. De ce preîncărcarea numărătorului în rutina de întrerupere introduce o eroare, iar ștergerea automată la comparare nu?
8. Un semnal PWM are perioada de 2 ms și durata stării active de 0,4 ms. Calculați frecvența și factorul de umplere. Dacă amplitudinea este de 5 V, care este tensiunea medie aplicată sarcinii și în ce condiții are sens să vorbim despre această valoare medie?
9. Explicați de ce comanda unui motor de curent continuu prin PWM are un randament mult mai bun decât reducerea tensiunii cu un rezistor serie. Ce rol joacă inductanța înfășurării și inerția rotorului?
10. Deduceți relația dintre frecvența semnalului PWM și rezoluția factorului de umplere, pentru un modul cu TOP programabil. Cât de multe trepte de reglaj se mai pot obține dacă se cere o frecvență PWM de 20 kHz de la un timer alimentat cu 16 MHz și prescalare 1?
11. Descrieți ce se întâmplă, la nivel de hardware, în momentul în care apare un front activ pe pinul de captură. De ce valoarea obținută astfel este mai exactă decât cea citită din numărător într-o rutină de întrerupere externă?
12. Pentru măsurarea duratei unui impuls care poate ajunge la 40 ms, cu Timer1 la 16 MHz, alegeți factorul de prescalare potrivit. Justificați alegerea prin calculul domeniului maxim măsurabil și al rezoluției obținute pentru fiecare variantă analizată.

7. Convertorul analog-numeric

De ce este nevoie de conversie analog-numerică
Caracteristicile de performanță ale unui convertor
Etapile procesului de conversie
Rata de eșantionare și fenomenul de aliasing
Rezoluția și eroarea de cuantizare
Tensiunea de referință
Timpul de conversie
Arhitecturi de implementare
Erori de conversie
Modulul ADC al microcontrolerului ATmega328P
Greșeli tipice și recomandări practice
Concluzii
Întrebări de verificare

7.1 De ce este nevoie de conversie analog-numerică

Toate mărimile pe care le măsurăm în lumea fizică sunt continue. Temperatura dintr-o încăpere nu sare de la douăzeci la douăzeci și unu de grade, ci trece prin toate valorile intermediare; presiunea într-un cilindru, poziția unghiulară a unui arbore, iluminarea unei suprafețe, tensiunea la bornele unui termorezistor - toate iau, cel puțin la scara la care ne interesează, o infinitate de valori, iar între oricare două valori distincte există întotdeauna o a treia. Un senzor este un dispozitiv care traduce o astfel de mărime într-o altă mărime continuă, de obicei o tensiune electrică, mai comodă de transportat și de prelucrat, dar la fel de continuă ca originalul.

Memoria unui microcontroler este exact opusul. Ea este alcătuită din celule binare, fiecare capabilă să rețină una din două stări, iar un grup de opt asemenea celule poate păstra exact 256 de configurații distincte, nici una în plus. Nu există, în interiorul unui sistem numeric, o cale de a memora „puțin mai mult decât 191 și puțin mai puțin decât 192”. Între lumea continuă a senzorului și lumea discretă a memoriei există așadar o incompatibilitate de principiu: informația trebuie *redușă* la un număr finit de valori, iar aceasta este o pierdere ireversibilă pe care proiectantul o acceptă deliberat și o dimensionează astfel încât să fie mai mică decât precizia care îl interesează.

Blocul care face această reducere se numește *convertor analog-numeric*, prescurtat ADC, după denumirea engleză *Analog to Digital Converter*. El primește la intrare o tensiune și livrează la ieșire un număr întreg, cuprins într-un domeniu fixat prin construcție, care reprezintă cât de mare a fost acea tensiune raportată la o tensiune de referință. Dacă, de exemplu, convertorul are o rezoluție de opt biți și lucrează cu o referință de 5 V, o tensiune de intrare de 3,742 V va fi transformată în numărul 191, pentru că 191 este cel mai apropiat multiplu întreg al treptei elementare de tensiune. Din acel moment, valoarea măsurată a devenit un simplu număr, pe care programul îl poate compara, aduna, filtra, transmite pe interfața serială sau înscrie într-o memorie.

Fără un asemenea bloc, un microcontroler ar putea comunica numai cu dispozitive care vorbesc deja limbajul lui: butoane, LED-uri, magistrale seriale. Toate traductoarele care redau o mărime fizică prin amplitudinea unei tensiuni - termistoare, fotorezistoare, potențiometre, traductoare de forță, microfoane, divizoare de măsurare a tensiunii unei baterii - ar rămâne în afara sistemului. Prezența unui ADC este ceea ce transformă un microcontroler dintr-un automat cu comenzi logice într-un instrument de măsură programabil.

Convertorul poate fi integrat în microcontroler, cum este cazul pe ATmega328P, sau poate fi un circuit extern legat printr-o magistrală serială. A doua variantă se alege atunci când se cere o rezoluție mai bună de zece-doisprezece biți sau o imunitate la zgomot pe care nu o poate oferi un bloc analogic aflat pe același siliciu cu un procesor care comută milioane de tranzistoare pe secundă.

7.2 Caracteristicile de performanță ale unui convertor

Înainte de a intra în mecanismul conversiei, este util să fixăm mărimile prin care se compară două convertoare, pentru că ele revin în tot restul capitolului și pentru că, în alegerea unui circuit, ele se află aproape întotdeauna în conflict: îmbunătățirea uneia se plătește prin degradarea alteia.

Rezoluția este numărul de biți pe care îi are rezultatul conversiei și determină în câte trepte distincte este împărțit domeniul de măsurare. Un convertor de r biți furnizează 2^r coduri diferite, adică 256 pentru opt biți, 1024 pentru zece și 4096 pentru doisprezece. Rezoluția nu spune nimic despre exactitatea măsurătorii, ci doar despre finețea ei; un convertor de șaisprezece biți prost proiectat poate fi mai puțin exact decât unul de zece biți bine făcut.

Lățimea de bandă este intervalul de frecvențe ale semnalului de intrare pe care convertorul le poate urmări fără distorsiuni semnificative. Ea nu este o proprietate independentă, ci decurge din rata de eșantionare: un convertor care produce zece mii de rezultate pe secundă nu poate reprezenta corect, așa cum vom vedea, decât semnale cu componente sub cinci kilohertzi.

Raportul semnal-zgomot, notat SNR, exprimă în decibeli raportul dintre puterea semnalului util și puterea zgomotului introdus de lanțul de măsurare, inclusiv de zgomotul propriu al cuantizării. Pentru un convertor ideal de r biți, atacat cu un semnal sinusoidal care ocupă tot domeniul, teoria dă valoarea $\text{SNR} = 6,02 \cdot r + 1,76$ dB, adică aproximativ 62 dB pen-

tru zece biți. Din formulă se reține mai ales observația că fiecare bit adăugat aduce cam șase decibeli; valoarea reală este întotdeauna mai mică, pentru că zgomotul de cuantizare se adaugă la cel al alimentării, al referinței și al sursei.

Timpul de conversie este intervalul dintre momentul declanșării unei conversii și momentul în care rezultatul este disponibil într-un registru. El limitează direct rata de eșantionare și, prin ea, lățimea de bandă.

Tensiunea de referință fixează domeniul de tensiuni pe care convertorul îl poate măsura. Ea intră în toate calculele de conversie, așa încât exactitatea ei se transmite integral în exactitatea rezultatelor.

Erorile specifice de conversie - de offset, de amplificare, de neliniaritate - descriu abaterea caracteristicii reale a convertorului de la caracteristica ideală și sunt tratate într-o secțiune separată.

Compromisul fundamental dintre aceste mărimi se enunță simplu: viteza, rezoluția și costul nu pot fi optimizate simultan. Un convertor foarte rapid este scump și are rezoluție mică; unul de rezoluție foarte mare este lent. Toată varietatea de arhitecturi discutată mai departe în acest capitol nu este altceva decât o colecție de compromisuri diferite între aceste trei mărimi.

7.3 Etapele procesului de conversie

Conversia unei tensiuni într-un număr nu este o operație elementară, ci un lanț de prelucrări care se succed într-o ordine obligatorie. Trei dintre ele sunt esențiale și dau structura oricărui convertor: *eșantionarea*, prin care semnalul continuu în timp este prelevat la momente discrete; *cuantizarea*, prin care valoarea prelevată, încă un număr real, este dusă la cel mai apropiat nivel dintr-o mulțime finită; și *codificarea cu stocare*, prin care nivelul ales este exprimat în binar și depus într-un registru. Primele două etape sunt independente una de alta: eșantionarea discretizează timpul, cuantizarea discretizează amplitudinea.

În jurul acestor trei etape se adaugă, într-un sistem real, încă două blocuri fără de care rezultatul ar fi lipsit de sens: un filtru trece-jos plasat înaintea eșantionatorului, care elimină din semnal componentele prea rapide pentru a fi reprezentate corect, și un circuit de reținere care imobilizează valoarea prelevată pe toată durata cuantizării. Lanțul complet este arătat în Figura 7.1.

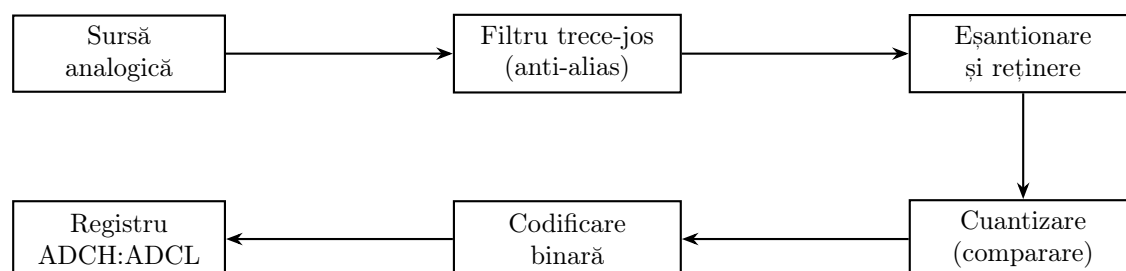


Figura 7.1 Lanțul complet de conversie analog-numerică. Filtrul limitează banda semnalului, circuitul de eșantionare și reținere îl imobilizează, blocul de cuantizare îl compară cu nivelele disponibile, iar rezultatul codificat ajunge într-un registru citibil de program.

7.3.1 Eșantionarea

A eșantiona înseamnă a măsura semnalul la momente de timp discrete, de regulă echidistante, și a păstra numai valorile obținute la aceste momente. Ceea ce se întâmplă între două eșantioane este pierdut definitiv; convertorul nu are cum să știe dacă semnalul a stat cuminte pe loc sau a făcut între timp o excursie amplă și s-a întors. Aceasta este prima și cea mai gravă pierdere de informație din tot lanțul, iar teorema eșantionării, discutată în secțiunea următoare, spune în ce condiții pierderea este totuși recuperabilă.

Se disting două moduri de a preleva. La *eșantionarea naturală*, comutatorul de intrare se închide pe o durată finită, iar valoarea transmisă mai departe urmărește în tot acest interval conturul semnalului. La *eșantionarea cu vârf plat*, valoarea captată în momentul închiderii comutatorului este menținută constantă pe toată durata impulsului, obținându-se o succesiune de paliere. Convertoarele reale folosesc a doua variantă, pentru un motiv care ține de etapa următoare: cuantizarea are nevoie de o valoare stabilă, nu de una care se mișcă.

7.3.2 Reținerea și circuitul de eșantionare și menținere

Blocul care realizează eșantionarea cu vârf plat se numește *circuit de eșantionare și reținere*, în engleză *sample and hold*. Structura lui este surprinzător de simplă: un comutator electronic, un condensator și un amplificator repetor. Când comutatorul se închide, condensatorul se încarcă la tensiunea de intrare; când se deschide, condensatorul rămâne cu sarcina acumulată, iar tensiunea la bornele lui, citită printr-un amplificator cu impedanță de intrare foarte mare, rămâne practic neschimbată. Faza cu comutatorul închis se numește *achiziție*, cea cu comutatorul deschis se numește *reținere*.

Necesitatea acestui bloc devine limpede dacă ne întrebăm ce s-ar întâmpla fără el. Un convertor cu aproximări succesive de zece biți, de tipul celui din ATmega328P, execută zece comparații succesive între tensiunea de intrare și o serie de tensiuni de probă. Fiecare comparație durează un ciclu de ceas al blocului de conversie, iar concluzia fiecăreia devine un bit al rezultatului. Dacă tensiunea de intrare s-ar modifica între prima și ultima comparație, deciziile luate ar privi valori diferite ale semnalului, iar numărul obținut la sfârșit nu ar corespunde nici uneia dintre ele. O variație de un singur pas de cuantizare petrecută la momentul nepotrivit poate falsifica bitul cel mai semnificativ, deci poate produce o eroare de jumătate din domeniul de măsurare. Concluzia este că, pe durata conversiei, tensiunea de pe condensatorul de eșantionare trebuie să rămână neschimbată; nu aproximativ neschimbată, ci neschimbată în limita a mai puțin de jumătate de pas de cuantizare.

Această cerință se traduce în două condiții practice. Prima privește *încărcarea* condensatorului: el trebuie să apuce să se încarce complet în intervalul scurt în care comutatorul este închis. Condensatorul de eșantionare al ATmega328P are aproximativ 14 pF, iar el se încarcă prin rezistența internă a comutatorului înseriată cu rezistența internă a sursei de semnal. Constanta de timp a acestui circuit este produsul celor două mărimi, iar pentru ca eroarea de încărcare să scadă sub jumătate de pas la zece biți trebuie să treacă aproximativ $\ln(2^{11}) \approx 7,6$ constante de timp. Cu o sursă de 10 k Ω , constanta de timp este

$10^4 \times 14 \cdot 10^{-12} = 140$ ns, deci încărcarea completă durează în jur de $1,1 \mu s$, mult mai puțin decât cele câteva microsecunde de achiziție disponibile. Cu o sursă de $1 M\Omega$, aceeași constantă devine $14 \mu s$, iar încărcarea ar avea nevoie de peste $100 \mu s$, adică de mai mult decât durează o conversie întreagă; rezultatul citit ar fi sistematic prea mic și, în plus, dependent de valoarea măsurată anterior pe alt canal. Aceasta este explicația fizică a recomandării din foaia de catalog ca impedanța sursei conectate la o intrare analogică să nu depășească $10 k\Omega$.

A doua condiție privește *descărcarea*: pe durata reținerii, condensatorul pierde inevitabil sarcină prin curenții de scurgere ai comutatorului și ai amplificatorului. Panta de coborâre a tensiunii, numită *droop*, trebuie să fie suficient de mică încât pierderea acumulată pe durata conversiei să rămână sub o fracțiune de pas de cuantizare. De aici rezultă o limitare care pare paradoxală: un convertor lent are nevoie de un condensator de reținere mai mare, dar un condensator mai mare se încarcă mai greu și limitează achiziția.

O variantă simplificată de reținere, întâlnită pe intrările analogice ale multor montaje, este pur și simplu un filtru trece-jos RC plasat la intrarea convertorului. El nu imobilizează semnalul, ci doar îl încetinește atât de mult încât variația lui pe durata conversiei devine neglijabilă; soluția este ieftină și rezolvă în același timp și problema anti-aliasului, dar introduce o întârziere.

7.3.3 Cuantizarea

Cuantizarea este operația prin care valoarea eșantionată, care este încă un număr real, este înlocuită cu cel mai apropiat element dintr-o mulțime finită de niveluri prestabilite. Domeniul de măsurare, cuprins între tensiunea de referință inferioară și cea superioară, este împărțit în intervale egale, iar fiecărui interval îi este atribuit un cod. Pentru un convertor de r biți există 2^r coduri, deci 2^r intervale, iar lățimea unui interval, numită *pas de cuantizare* sau *tensiune a bitului cel mai puțin semnificativ* (LSB), este

$$q = \frac{V_{REF+} - V_{REF-}}{2^r}.$$

Figura 7.2 arată efectul cuantizării. Caracteristica ideală de transfer ar fi dreapta continuă; cea reală este scara, care rămâne constantă pe lățimea unui interval și sare cu o unitate la trecerea în intervalul următor. Diferența dintre ele, măsurată pe verticală, este *eroarea de cuantizare*.

Eroarea de cuantizare este, spre deosebire de celelalte erori discutate în acest capitol, o eroare de principiu: ea nu provine dintr-un defect de fabricație și nu poate fi eliminată prin calibrare, ci este consecința inevitabilă a înlocuirii unei mulțimi infinite cu una finită, iar singurul mod de a o reduce este creșterea numărului de biți. Dacă rotunjirea se face către cel mai apropiat nivel, eroarea este cuprinsă între $-q/2$ și $+q/2$, deci nu depășește jumătate de pas. Multe convertoare reale, printre care și cel din ATmega328P, efectuează în schimb o trunchiere: codul returnat este partea întreagă a raportului dintre tensiunea de intrare și pas, astfel încât eroarea este cuprinsă între 0 și q și are întotdeauna același sens. Diferența reprezintă un offset sistematic de jumătate de pas, compensabil în software.

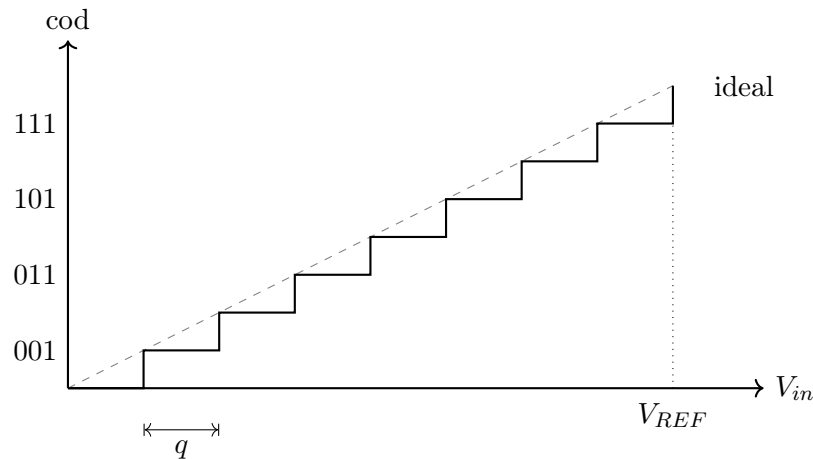


Figura 7.2 Caracteristica de transfer a unui cuantizor de 3 biți, suprapusă peste dreapta ideală. Fiecare treaptă are lățimea unui pas de cuantizare q , iar abaterea pe verticală dintre scară și dreaptă este eroarea de cuantizare.

7.3.4 Codificarea și stocarea

Ultima etapă este cea mai simplă, dar fără ea rezultatul nu ar fi accesibil programului. Nivelul de cuantizare determinat anterior este exprimat ca număr binar, în codificare naturală pentru convertoarele unipolare sau într-o codificare cu semn pentru cele bipolare, iar numărul obținut este înscris într-un registru de rezultat al perifericului, de unde poate fi citit de unitatea centrală printr-o instrucțiune obișnuită.

Pe ATmega328P, unde rezultatul are zece biți, registrul de rezultat este format din două registre de opt biți, ADCH și ADCL. Simultan cu depunerea rezultatului, hardware-ul activează un indicator de stare care semnalează terminarea conversiei și, dacă a fost validată, o întrerupere. Din acel moment convertorul este liber pentru o nouă conversie, iar valoarea rămâne în registru până la suprascrierea ei de rezultatul următor; dacă programul întârzie prea mult, o valoare poate fi pierdută fără niciun avertisment.

7.4 Rata de eșantionare și fenomenul de aliasing

7.4.1 Rata de eșantionare

Rata de eșantionare este numărul de eșantioane prelevate într-o secundă, măsurată în hertzi sau, mai sugestiv, în eșantioane pe secundă. Ea este mărimea inversă a intervalului dintre două prelevări succesive și, atunci când convertorul lucrează continuu, este limitată de timpul de conversie:

$$R_e = \frac{1}{T_c}.$$

Relația arată de ce timpul de conversie contează atât de mult: el fixează plafonul până la care se poate ridica rata de eșantionare și, prin ea, banda de frecvențe pe care sistemul o poate urmări. Rata efectivă este de obicei mai mică decât acest plafon, pentru că între conversii programul trebuie să citească rezultatul și, eventual, să schimbe canalul multiplexorului.

7.4.2 Teorema eşantionării

Întrebarea fundamentală a eşantionării este cât de des trebuie să privim un semnal pentru a-l putea reconstitui. Răspunsul, dat de teorema eşantionării, cunoscută și sub numele de teorema lui Shannon sau criteriul lui Nyquist, este că frecvența de eşantionare trebuie să depășească strict dublul celei mai înalte frecvențe prezente în semnal:

$$f_s > 2f_{max},$$

sau, în formă echivalentă,

$$f_{max} < \frac{f_s}{2}.$$

Frecvența $f_s/2$ se numește *frecvență Nyquist* și reprezintă limita superioară a benzii pe care sistemul o poate reprezenta corect.

Motivul intuitiv al factorului doi este că o oscilație are, în fiecare perioadă, un maxim și un minim, iar pentru a o distinge de o linie dreaptă este nevoie de cel puțin câte o observație în vecinătatea fiecăruia. Cu mai puțin de două eşantioane pe perioadă, aceleași puncte pot fi explicate la fel de bine printr-o oscilație mai lentă. Teorema cere, de altfel, o inegalitate strictă: la exact două eşantioane pe perioadă, dacă prelevările cad la trecerile prin zero, toate eşantioanele sunt nule și semnalul dispăre complet.

7.4.3 Aliasing

Atunci când condiția teoremei nu este respectată, componentele de frecvență prea înaltă nu se pierd pur și simplu, ci se transformă în componente de frecvență joasă, care apar în rezultat ca și cum ar fi făcut dintotdeauna parte din semnal. Fenomenul se numește *aliasing*, adică luare a unei frecvențe drept alta, iar frecvența aparentă se obține prin coborârea repetată a frecvenței reale cu câte un multiplu al frecvenței de eşantionare, până când rezultatul cade în banda utilă:

$$f_{alias} = \min_k |f_{in} - kf_s|,$$

unde k parcurge numerele întregi.

Exemplu

Un semnal sinusoidal de 6 kHz este eşantionat cu o frecvență de 8 kHz. Ce frecvență va avea semnalul reconstituit din eşantioane?

Rezolvare. Frecvența Nyquist este $f_s/2 = 4$ kHz, iar semnalul are 6 kHz, deci condiția teoremei este încălcată. Aplicând formula pentru $k = 1$ se obține

$$f_{alias} = |6 - 8| = 2 \text{ kHz},$$

iar celelalte valori ale lui k dau rezultate mai mari (14 kHz pentru $k = -1$, 10 kHz pentru $k = 2$), deci frecvența aparentă este 2 kHz.

Verificarea directă este instructivă. Eșantionul de rang k al sinusoidei de 6 kHz este

$$x[k] = \sin\left(2\pi \cdot 6000 \cdot \frac{k}{8000}\right) = \sin\left(\frac{3\pi k}{2}\right),$$

iar cel al unei sinusoide de 2 kHz este $\sin(\pi k/2)$. Scăzând $2\pi k$ din argumentul primei expresii, ceea ce nu schimbă valoarea sinusului, se obține $\sin(-\pi k/2) = -\sin(\pi k/2)$. Cele două șiruri de eșantioane sunt așadar identice până la semn: pentru $k = 0, 1, 2, 3, 4$ semnalul de 6 kHz dă valorile $0, -1, 0, +1, 0$, iar cel de 2 kHz dă $0, +1, 0, -1, 0$.

Concluzia este că, privind numai eșantioanele, este imposibil de decis dacă la intrare a fost o sinusoidă de 6 kHz sau una de 2 kHz cu faza inversată. Componenta de 6 kHz nu a dispărut: ea s-a mutat, ca un intrus, în mijlocul benzii utile, unde niciun filtru numeric nu o mai poate deosebi de un semnal legitim.

Consecințele aliasingului sunt trei, și toate sunt grave: semnalul reconstituit nu mai reprezintă originalul, deci măsurătoarea este falsă; informația din banda înaltă este pierdută, fiind suprapusă peste banda joasă; iar operația nu este inversabilă, pentru că din datele eșantionate nu se mai poate separa semnalul util de intrus, oricât de sofisticată ar fi prelucrarea ulterioară.

Observație

Aliasingul nu poate fi corectat prin software după eșantionare. Odată ce o componentă de frecvență înaltă a fost coborâtă în banda utilă, ea este indistinctibilă de un semnal legitim. Singurele remedii sunt preventive: creșterea frecvenței de eșantionare și, mai ales, un filtru trece-jos *analogic* plasat înaintea convertorului.

7.4.4 Filtrul anti-alias

Din faptul că aliasingul este ireversibil rezultă că singura apărare eficace este să nu lăsăm componentele periculoase să ajungă la eșantionator. Aceasta este misiunea *filtrului anti-alias*: un filtru trece-jos analogic, plasat între sursa de semnal și intrarea convertorului, cu frecvența de tăiere sub frecvența Nyquist. Faptul că filtrul trebuie să fie analogic nu este o preferință, ci o necesitate logică: el trebuie să acționeze înainte de eșantionare, iar orice prelucrare numerică vine, prin definiție, după.

În forma cea mai simplă, filtrul este o celulă RC, cu un rezistor în serie și un condensator la masă, a cărei frecvență de tăiere este

$$f_t = \frac{1}{2\pi RC}.$$

O asemenea celulă atenuează cu 20 dB pe decadă, adică destul de lent, motiv pentru care frecvența ei de tăiere se alege sensibil sub frecvența Nyquist, iar frecvența de eșantionare se ia cu o marjă confortabilă peste dublul benzii utile - în practică, de trei până la zece ori banda utilă, nu de exact două ori. Această supradimensionare, numită supraeșantionare, are

un preț mic în calcul și un câștig mare în siguranță. Acolo unde marja nu poate fi luată, se folosesc filtre active de ordin superior, cu pantă mult mai abruptă.

Alegerea rezistorului din filtru trebuie făcută însă cu grijă, pentru că el se adaugă la impedanța sursei văzută de condensatorul de eșantionare: un filtru cu $R = 100 \text{ k}\Omega$ rezolvă problema aliasingului și o creează pe cea a încărcării incomplete. Combinația uzuală este un rezistor de câțiva kilohmi cu un condensator de ordinul nanofarazilor.

7.5 Rezoluția și eroarea de cuantizare

Rezoluția unui convertor se exprimă în două feluri, care nu trebuie confundate. Prima este rezoluția *în biți*, adică numărul r de cifre binare ale rezultatului, din care rezultă numărul de stări distincte 2^r . A doua este rezoluția *în volți*, adică valoarea în tensiune a unei singure trepte, care depinde atât de numărul de biți, cât și de domeniul de măsurare.

Fie E_{mt} diferența dintre tensiunea maximă și tensiunea minimă pe care convertorul le poate măsura, adică $V_{REF+} - V_{REF-}$. Pasul de cuantizare, a cărui valoare este chiar tensiunea corespunzătoare bitului cel mai puțin semnificativ, este

$$q = \frac{E_{mt}}{2^r}.$$

În literatură se întâlnește frecvent și formula cu numitorul $2^r - 1$. Cele două nu sunt în contradicție, ci privesc lucruri diferite: $E_{mt}/2^r$ este lățimea unui interval de cuantizare, în timp ce $E_{mt}/(2^r - 1)$ este distanța dintre două coduri consecutive atunci când se cere ca ultimul cod să corespundă exact tensiunii maxime, adică atunci când codurile sunt privite ca puncte, nu ca intervale. Pentru ATmega328P este valabilă prima convenție: codul 0 corespunde masei analogice, iar codul maxim 1023 corespunde referinței minus un pas. Diferența numerică este sub o miime la zece biți, deci fără consecințe practice.

Exemplu

Un convertor de 10 biți lucrează cu $V_{REF-} = 0$ și $V_{REF+} = 5 \text{ V}$. Care este pasul de cuantizare, care este eroarea maximă de măsurare datorată numai cuantizării și ce cod se obține pentru o tensiune de intrare de 3,742 V?

Rezolvare. Numărul de niveluri este $2^{10} = 1024$, deci pasul de cuantizare este

$$q = \frac{5 \text{ V}}{1024} = 4,8828 \cdot 10^{-3} \text{ V} \approx 4,9 \text{ mV}.$$

Aceasta este cea mai mică variație de tensiune care mai poate produce o schimbare a codului de ieșire; orice variație mai mică trece neobservată.

Dacă rotunjirea s-ar face la cel mai apropiat nivel, eroarea de cuantizare nu ar depăși $q/2 \approx 2,44 \text{ mV}$. Convertorul din ATmega328P trunchiază, deci eroarea este cuprinsă între 0 și 4,9 mV, cu o valoare medie de 2,44 mV, care este un offset sistematic.

Codul corespunzător tensiunii de 3,742 V se obține din raportul

$$\frac{V_{in}}{q} = \frac{3,742}{4,8828 \cdot 10^{-3}} = 766,3,$$

din care, prin trunchiere, rezultă codul 766. Tensiunea reconstituită din acest cod este $766 \times 4,8828 \text{ mV} = 3,740 \text{ V}$, deci abaterea este de 2 mV, într-adevăr mai mică de un pas. O observație finală: dacă senzorul care furnizează cei 3,742 V are o eroare proprie de 1 %, adică de aproximativ 37 mV, atunci rezoluția de 4,9 mV a convertorului este de opt ori mai fină decât precizia semnalului măsurat. Creșterea rezoluției la doisprezece biți nu ar aduce, într-un astfel de caz, niciun câștig real de exactitate.

Exemplul de mai sus conține lecția cea mai importantă despre rezoluție: ea trebuie aleasă în raport cu variația utilă a mărimii măsurate, nu maximizată de dragul cifrei. Dacă un termocuplu produce $40 \mu\text{V}$ pe grad Celsius, un convertor de zece biți cu referință de 5 V, la care un pas valorează 4,9 mV, nu poate distinge nici măcar o sută de grade; soluția nu este un convertor cu mai mulți biți, ci un amplificator care aduce semnalul la scara domeniului de măsurare.

7.6 Tensiunea de referință

Tensiunea de referință este mărimea cu care se compară, direct sau indirect, tensiunea de măsurat. Ea este dată prin două valori, V_{REF-} și V_{REF+} , care delimitează domeniul de conversie: o tensiune de intrare egală cu V_{REF-} produce codul zero, iar una egală cu V_{REF+} produce, prin convenție, codul maxim $2^r - 1$. Între aceste limite, corespondența este liniară, iar codul obținut pentru o tensiune de intrare V_{in} este

$$N = \left\lfloor 2^r \cdot \frac{V_{in} - V_{REF-}}{V_{REF+} - V_{REF-}} \right\rfloor.$$

Relația inversă, care dă tensiunea corespunzătoare unui cod N citit din registru, este

$$V_{in} = V_{REF-} + N \cdot \frac{V_{REF+} - V_{REF-}}{2^r}.$$

A doua formulă este cea folosită în programe, imediat după citirea rezultatului, și trebuie să devină un automatism.

Exemplu

Un convertor de 10 biți lucrează între $V_{REF-} = 0 \text{ V}$ și $V_{REF+} = 5 \text{ V}$. Ce tensiune de intrare corespunde codului $N = 100$?

Rezolvare. Aplicând relația inversă,

$$V_{in} = 0 + 100 \cdot \frac{5 - 0}{1024} = 100 \times 4,8828 \text{ mV} = 488,3 \text{ mV} \approx 0,49 \text{ V}.$$

Rezultatul se poate verifica și prin raționamentul invers: codul 100 din 1024 reprezintă $100/1024 \approx 9,77\%$ din domeniu, iar $9,77\%$ din 5 V înseamnă 0,49 V.

Este de reținut că valoarea obținută nu este exactă, ci reprezintă limita inferioară a intervalului de tensiuni care produc codul 100. Toate tensiunile de intrare cuprinse între 488,3 mV și 493,2 mV dau același rezultat.

7.6.1 Sursele de referință ale microcontrolerului ATmega328P

Convertorul din ATmega328P are V_{REF-} legat permanent la masa analogică, iar pentru V_{REF+} oferă trei posibilități, selectate prin biții REFS1 și REFS0 din registrul ADMUX, prezentate în Tabelul 7.1.

Tabela 7.1 Sursele de tensiune de referință ale convertorului din ATmega328P

REFS1	REFS0	Sursă	Observații
0	0	Pinul AREF extern	Referința internă este deconectată; domeniul este fixat de tensiunea aplicată din exterior
0	1	AVCC	Tensiunea de alimentare analogică, cu condensator de decuplare la AREF; este varianta implicită a bibliotecii Arduino
1	1	Referința internă de 1,1 V	Independentă de alimentare, dar cu dispersie de ordinul $\pm 0,1$ V de la exemplar la exemplar

Alegerea între aceste variante nu este o formalitate, pentru că ea fixează simultan două lucruri: domeniul de măsurare și rezoluția efectivă. Cu referința pe AVCC, adică pe 5 V, se pot măsura tensiuni de la 0 la 5 V cu un pas de 4,9 mV. Cu referința internă de 1,1 V, domeniul se restrânge la 0-1,1 V, dar pasul devine $1,1/1024 \approx 1,07$ mV, deci de aproape cinci ori mai fin. Un semnal care nu depășește niciodată o jumătate de volt va fi măsurat mult mai bine cu referința internă; același semnal, măsurat cu referința pe 5 V, ar folosi numai o zecime din codurile disponibile, adică ar fi reprezentat efectiv pe puțin mai mult de șapte biți din cei zece. Regula practică este să se aleagă cea mai mică referință care mai cuprinde întregul domeniu util al semnalului, cu o marjă rezonabilă.

Referința internă are avantajul de a fi independentă de tensiunea de alimentare, ceea ce o face potrivită pentru montaje alimentate din baterie. În schimb, valoarea ei nominală de 1,1 V nu este garantată exact, foaia de catalog admitând o dispersie de ordinul zecimii de volt între exemplare; într-o măsurătoare absolută, aceasta se traduce într-o eroare de amplificare de până la zece procente, eliminabilă numai prin calibrare individuală.

Cazul cel mai supărător este însă referința pe AVCC, tocmai pentru că este cea implicită și pentru că pare inofensivă. Ea presupune că tensiunea de alimentare este de exact 5 V, ceea ce este rareori adevărat. O placă Arduino alimentată prin USB primește, după dioda de protecție și după toleranțele portului gazdă, o tensiune care poate fi de 4,6-4,8 V; aceeași placă alimentată prin stabilizatorul propriu are aproximativ 5,0 V. Programul care calculează tensiunea împărțind codul la 1024 și înmulțind cu 5 va da, în primul caz, rezultate cu patru-cinci procente mai mari decât realitatea, fără nici un semn de eroare. Mai mult, dacă alimentarea variază în timp - de exemplu pentru că montajul comandă un motor care trage curent în impulsuri - referința variază odată cu ea, iar rezultatele măsurătorilor „respiră” în ritmul consumului.

Există totuși o categorie de măsurători în care referința pe AVCC este alegerea corectă și chiar avantajoasă: măsurătorile *rationetrice*. Dacă senzorul este alimentat din aceeași tensiune care servește drept referință, cum se întâmplă la un potențiomtru conectat între V_{CC} și masă sau la un divizor rezistiv, atunci tensiunea furnizată de senzor este o fracțiune fixă din alimentare, iar raportul dintre ea și referință nu depinde de valoarea alimentării. Codul citit rămâne corect chiar dacă alimentarea este de 4,7 V în loc de 5 V, pentru că ambele mărimi se modifică proporțional și eroarea se simplifică.

Observație

Dacă biții REFS selectează AVCC sau referința internă, la pinul AREF apare tensiunea de referință generată intern. Conectarea unei surse externe la AREF în această situație scurtcircuitază cele două referințe și poate distruge circuitul. Pinul AREF se leagă, în funcționare normală, numai printr-un condensator de decuplare la masă.

O ultimă observație privește calitatea referinței. Orice zgomot suprapus peste tensiunea de referință se regăsește în toate rezultatele măsurate, nu numai în cele de pe un canal, ci în toate deodată, inclusiv în cele care par cel mai stabile. Un vârf de zgomot de 50 mV pe o referință de 5 V reprezintă un procent din domeniu, adică zece pași de cuantizare la zece biți, deci face inutili ultimii trei biți ai rezultatului. De aceea foaia de catalog cere un condensator de decuplare la pinul AREF și, pentru măsurători serioase, alimentarea părții analogice printr-un filtru propriu.

7.7 Timpul de conversie

Timpul de conversie este intervalul dintre declanșarea unei conversii și momentul în care rezultatul devine disponibil în registru. El depinde de trei factori: de arhitectura convertorului, care fixează câte operații elementare sunt necesare; de rezoluție, pentru că un convertor cu aproximații succesive are nevoie de câte un ciclu pentru fiecare bit; și de frecvența ceasului care ritmează blocul de conversie. Într-o formă generală, aplicabilă convertoarelor secvențiale, timpul de conversie se scrie

$$T_c = \tau_{ADC} \cdot r + \alpha \cdot \tau_{sistem},$$

unde τ_{ADC} este durata unei operații elementare, adică perioada ceasului de conversie, r este numărul de biți, iar termenul al doilea adună toate duratele fixe care nu depind de rezoluție: achiziția eșantionului, sincronizarea cu ceasul sistemului, transferul rezultatului în registru. Constanta α depinde de implementare și se citește din foaia de catalog.

7.7.1 Ceasul convertorului din ATmega328P

Blocul de conversie al ATmega328P nu lucrează cu ceasul sistemului, ci cu un ceas propriu, obținut din acesta printr-un circuit de prescalare comandat de biții ADPS2:0 din registrul ADCSRA. Motivul acestei separări este fizic: comparatorul intern și rețeaua de condensatoare care generează tensiunile de probă au nevoie de un timp minim pentru a se stabili, iar acest timp nu se scurtează oricât de repede ar merge procesorul. Foaia de catalog cere ca ceasul convertorului să se afle între 50 și 200 kHz pentru a se obține rezoluția completă de zece biți; peste 200 kHz convertorul continuă să funcționeze, dar ultimii biți ai rezultatului devin nesiguri, ceea ce echivalează cu pierderea de rezoluție.

O conversie obișnuită durează 13 cicluri ale acestui ceas: unul și jumătate pentru achiziția eșantionului, zece pentru cele zece comparații succesive și restul pentru sincronizare și pentru scrierea rezultatului. Prima conversie de după activarea modulului este mai lungă, 25 de cicluri, pentru că include și pornirea circuitului analogic.

Tabela 7.2 Factorii de prescalare ai ceasului ADC și timpul de conversie rezultat, pentru un ceas de sistem de 16 MHz

ADPS2:0	Factor	Ceas ADC	Conversie (13 cicluri)	În domeniul recomandat
000, 001	2	8 MHz	1,63 μ s	nu
010	4	4 MHz	3,25 μ s	nu
011	8	2 MHz	6,5 μ s	nu
100	16	1 MHz	13 μ s	nu
101	32	500 kHz	26 μ s	nu
110	64	250 kHz	52 μ s	nu
111	128	125 kHz	104 μ s	da

Tabelul 7.2 arată de ce, pe o placă Arduino Uno cu ceas de 16 MHz, singura alegere care respectă condiția de rezoluție completă este factorul de prescalare 128. Calculul este direct:

$$f_{ADC} = \frac{16 \cdot 10^6}{128} = 125 \text{ kHz}, \quad T_c = \frac{13}{125 \cdot 10^3} = 1,04 \cdot 10^{-4} \text{ s} = 104 \mu\text{s}.$$

Aceasta este exact valoarea pe care o are, în practică, un apel `analogRead()`, iar rata de eșantionare corespunzătoare este

$$R_e = \frac{1}{104 \mu\text{s}} \approx 9600,$$

adică aproximativ 9600 de eșantioane pe secundă.

Consecința asupra benzii utile se obține imediat din teorema eșantionării: frecvența Nyquist este de aproximativ 4,8 kHz, deci semnalul măsurat nu trebuie să conțină componente peste această valoare. Ținând seama de marja necesară filtrului anti-alias, banda utilă realistă a unui montaj Arduino care citește continuu o intrare analogică este de ordinul unui kilohertz sau doi - suficient pentru temperaturi, poziții sau tensiuni de baterie, insuficient pentru audio.

Dacă aplicația cere o rată mai mare, se poate coborî factorul de prescalare la 64, acceptând conștient pierderea ultimilor biți: la 250 kHz timpul de conversie scade la $52 \mu\text{s}$, deci rata se dublează, iar rezultatul rămâne utilizabil pe opt-nouă biți. Este un compromis legitim atât timp cât este declarat; ce nu se poate face este să se ceară simultan viteză mare și zece biți onești.

Exemplu

Un montaj trebuie să eșantioneze alternativ două canale analogice, cu aceeași frecvență pe fiecare canal, folosind convertorul unui ATmega328P la 16 MHz cu factorul de prescalare 128. Care este frecvența maximă de eșantionare pe canal și care este banda utilă a fiecărui semnal?

Rezolvare. Convertorul este unic și multiplexat, deci cele două canale își împart aceeași capacitate de conversie. Un ciclu complet de citire a ambelor canale durează cel puțin $2 \times 104 = 208 \mu\text{s}$, la care se adaugă timpul de achiziție după fiecare comutare a multiplexorului și timpul consumat de program. Frecvența de eșantionare pe canal este deci

$$f_s = \frac{1}{208 \mu\text{s}} \approx 4800 \text{ Hz},$$

iar frecvența Nyquist corespunzătoare este de aproximativ 2,4 kHz. Cu o marjă rezonabilă pentru filtrul anti-alias, fiecare dintre cele două semnale nu ar trebui să conțină componente peste circa 1 kHz.

Observăm astfel un efect care surprinde adesea la prima aplicație cu mai multe canale: adăugarea unui al doilea senzor înjumătățește banda utilă a primului, pentru că resursa partajată nu este pinul, ci blocul de conversie.

7.8 Arhitecturi de implementare

Toate convertoarele fac, în esență, același lucru: compară tensiunea de măsurat cu una sau mai multe tensiuni cunoscute și deduc din rezultatul comparațiilor un număr. Ceea ce le deosebește este *cum* organizează aceste comparații - toate deodată, una câte una, sau prin numărare - iar de aici decurg diferențele de viteză, de rezoluție, de consum și de cost.

7.8.1 Comparatorul, convertorul de un bit

Elementul comun tuturor arhitecturilor este *comparatorul*, un circuit cu două intrări analogice și o ieșire numerică, a cărei ieșire indică pur și simplu care dintre cele două tensiuni

de intrare este mai mare:

$$\begin{cases} V_{in-} > V_{in+} & \Rightarrow \text{Out} = 1 \\ V_{in-} \leq V_{in+} & \Rightarrow \text{Out} = 0 \end{cases}$$

Un comparator este, cu deplină îndreptăţire, un convertor analog-numeric de un singur bit: el reduce o mărime continuă la două stări, folosind un prag drept referinţă. ATmega328P dispune de un asemenea comparator analogic independent de convertor, care poate genera o întrerupere la schimbarea ieşirii şi care este util pentru detecţia pragurilor, fără a consuma timpul unei conversii complete.

7.8.2 Convertorul cu aproximări succesive

Arhitectura cea mai răspândită în microcontrolere, şi cea folosită de ATmega328P, este cea cu *aproximări succesive*, prescurtată SAR după denumirea engleză *Successive Approximation Register*. Blocurile componente sunt patru: un registru de aproximări succesive, care păstrează codul construit până în acel moment; un convertor numeric-analogic care transformă conţinutul acestui registru într-o tensiune de probă; o referinţă care fixează scara acestui convertor; şi un comparator care compară tensiunea de probă cu tensiunea de măsurat. Rezultatul final este copiat într-un registru de memorie accesibil programului.

Algoritmul este o *căutare binară pe tensiune*, identică în principiu cu metoda prin care se ghiceşte un număr punând întrebări de tipul „este mai mare decât ...?”. Se porneşte cu toţi biţii pe zero, se pune pe 1 bitul cel mai semnificativ, ceea ce generează o tensiune de probă egală cu jumătate din domeniu, şi se întreabă comparatorul dacă tensiunea de măsurat este mai mare. Dacă răspunsul este da, bitul rămâne pe 1 şi valoarea se află în jumătatea superioară a domeniului; dacă răspunsul este nu, bitul este şters, iar valoarea se află în jumătatea inferioară. Se trece apoi la bitul următor, care împarte în două jumătatea rămasă, şi se repetă raţionamentul. După r paşi, intervalul de incertitudine s-a redus de 2^r ori, iar codul acumulat este rezultatul.

Eficienţa metodei vine tocmai din această înjumătăţire: fiecare comparaţie elimină exact jumătate din posibilităţile rămase, deci numărul de comparaţii necesare este logaritmul în baza doi al numărului de coduri, adică exact numărul de biţi. Un convertor de zece biţi are nevoie de zece comparaţii, nu de o mie douăzeci şi patru, cum ar cere o căutare exhaustivă.

Figura 7.3 reprezintă acest proces ca pe un arbore de decizie, pentru un convertor de trei biţi cu referinţa de 5 V. Fiecare nod conţine tensiunea de probă generată la pasul respectiv, fiecare ramură corespunde unui răspuns al comparatorului, iar frunza atinsă după trei decizii este codul rezultat. Traseul îngroşat urmăreşte conversia unei tensiuni de intrare de 3,6 V.

Exemplu

Să se urmărească pas cu pas conversia tensiunii $V_{in} = 3,6$ V de către un convertor cu aproximări succesive de 4 biţi, cu $V_{REF-} = 0$ şi $V_{REF+} = 5$ V.

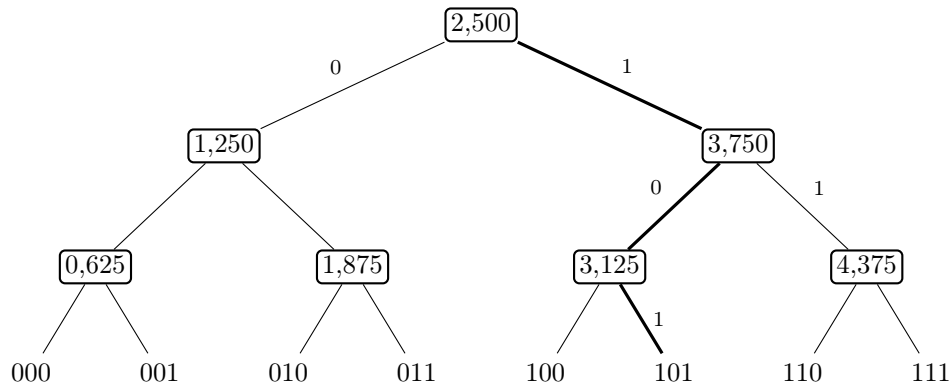


Figura 7.3 Arborele de decizie al unui convertor cu aproximări succesive de 3 biți, cu referința de 5 V. Nodurile conțin tensiunile de probă, exprimate în volți; traseul îngroșat corespunde conversiei tensiunii de intrare de 3,6 V, care produce codul 101.

Rezolvare. Pasul de cuantizare este

$$q = \frac{5}{2^4} = \frac{5}{16} = 0,3125 \text{ V.}$$

Registrul pornește de la valoarea 0000. La fiecare pas, bitul curent este pus provizoriu pe 1, se calculează tensiunea de probă ca produs dintre valoarea zecimală a registrului și pasul q , iar comparatorul decide dacă bitul rămâne sau nu.

Pasul 1, bitul 3 (MSB). Cod de probă 1000, adică 8, deci tensiunea de probă este $8 \times 0,3125 = 2,5 \text{ V}$. Cum $3,6 > 2,5$, bitul rămâne pe 1. Registrul devine 1000, iar valoarea se știe acum a fi în intervalul $[2,5; 5] \text{ V}$.

Pasul 2, bitul 2. Cod de probă 1100, adică 12, deci tensiunea de probă este $12 \times 0,3125 = 3,75 \text{ V}$. Cum $3,6 < 3,75$, bitul este șters. Registrul rămâne 1000, iar intervalul se restrânge la $[2,5; 3,75] \text{ V}$.

Pasul 3, bitul 1. Cod de probă 1010, adică 10, deci tensiunea de probă este $10 \times 0,3125 = 3,125 \text{ V}$. Cum $3,6 > 3,125$, bitul rămâne. Registrul devine 1010, iar intervalul este $[3,125; 3,75] \text{ V}$.

Pasul 4, bitul 0 (LSB). Cod de probă 1011, adică 11, deci tensiunea de probă este $11 \times 0,3125 = 3,4375 \text{ V}$. Cum $3,6 > 3,4375$, bitul rămâne. Registrul final este 1011, adică 11 în zecimal.

Tensiunea reconstituită din acest cod este $11 \times 0,3125 = 3,4375 \text{ V}$, deci eroarea de conversie este $3,6 - 3,4375 = 0,1625 \text{ V}$, adică $0,52 \cdot q$, mai puțin de un pas de cuantizare, așa cum era de așteptat de la o conversie prin trunchiere.

Se observă că intervalul de incertitudine s-a înjumătățit la fiecare pas: de la 5 V inițial la 2,5, apoi 1,25, apoi 0,625 și în final 0,3125 V, adică exact un pas de cuantizare. Se observă de asemenea că primele trei decizii au dat 101, adică exact rezultatul conversiei pe trei biți din Figura 7.3: biții unui convertor cu aproximări succesive se obțin în ordinea importanței lor, iar oprirea prematură a procesului dă rezultatul corect pe un număr mai

mic de biți.

Această ultimă proprietate explică și de ce arhitectura SAR este atât de potrivită pentru microcontrolere. Timpul de conversie este proporțional cu numărul de biți, deci previzibil și constant indiferent de valoarea măsurată; consumul este redus, pentru că există un singur comparator activ; iar rezoluția de zece-doisprezece biți, obținută la câteva zeci sau sute de mii de eșantioane pe secundă, acoperă exact nevoile aplicațiilor de măsurare uzuale. Punctul slab este dependența de circuitul de eșantionare și reținere, care complică proiectarea.

7.8.3 Convertorul paralel (flash)

La polul opus se află *convertorul paralel*, cunoscut mai ales sub numele englezesc de *flash*. Ideea lui este să nu facă nicio căutare, ci să compare tensiunea de intrare, simultan, cu toate nivelele de cuantizare posibile. Un divizor rezistiv format din rezistoare egale, alimentat de la tensiunea de referință, generează toate pragurile intermediare; fiecare prag atacă intrarea inversoare a câte unui comparator, iar toate intrările neinverse sunt legate împreună la semnalul de măsurat. Ieșirile comparatoarelor formează un cod „termometric”: toate comparatoarele ale căror praguri sunt sub tensiunea de intrare dau 1, restul dau 0, iar poziția tranziției indică valoarea. Un codificator cu priorități transformă acest cod termometric în cod binar obișnuit.

De exemplu, un convertor paralel de trei biți cu referința de 8 V are pragurile la 1, 2, ..., 7 V. O tensiune de intrare de 3,3 V activează primele trei comparatoare, iar codificatorul cu priorități reține poziția cea mai înaltă activă, adică trei, și generează codul binar 011.

Avantajul este viteza: conversia se face într-un singur pas, limitat doar de timpul de răspuns al comparatoarelor, ceea ce permite rate de miliarde de eșantioane pe secundă. Dezavantajul este costul, în sensul cel mai concret: un convertor de r biți are nevoie de $2^r - 1$ comparatoare, adică 255 pentru opt biți, 1023 pentru zece și peste patru mii pentru doispzezece. Numărul crește exponențial cu rezoluția, iar odată cu el cresc suprafața de siliciu, consumul și capacitatea parazită de la intrare. De aceea convertoarele paralele se opresc de regulă la opt biți și se folosesc acolo unde viteza este singurul criteriu: osciloscops digitale, receptoare radio, achiziție video.

7.8.4 Convertorul cu rampă

O a treia familie renunță complet la viteză în favoarea simplității și a exactității. *Convertorul cu rampă* pornește simultan un generator de tensiune liniar crescătoare, obținut prin încărcarea la curent constant a unui condensator, și un numărător comandat de un ceas stabil. În momentul în care rampa depășește tensiunea de intrare, comparatorul oprește numărătorul, iar valoarea acumulată în acesta este proporțională cu tensiunea măsurată, pentru că timpul necesar rampei ca să atingă un anumit nivel este proporțional cu acel nivel. Timpul de conversie nu este constant, ci depinde de valoarea măsurată, iar în cazul cel mai defavorabil înseamnă 2^r perioade de ceas, adică de o sută de ori mai mult decât la un convertor cu aproximări succesive de aceeași rezoluție.

Varianta perfecționată, numită *convertor cu dublă rampă*, integrează mai întâi tensiunea de intrare pe un interval de timp fix, apoi descarcă integratorul cu o tensiune de referință de semn opus și măsoară timpul necesar descărcării. Trucul acestei metode este că valoarea condensatorului și frecvența ceasului intervin în ambele faze și se simplifică din raport, astfel încât rezultatul nu depinde de ele, ci numai de referință și de raportul celor doi timpi. În plus, integrarea pe un interval fix mediază semnalul, iar dacă intervalul este ales multiplu al perioadei rețelei de alimentare, perturbația de 50 Hz se anulează exact. Aceste două proprietăți fac din convertorul cu dublă rampă soluția clasică a multimetrelor digitale, unde se cer patru-cinci cifre exacte, iar câteva măsurători pe secundă sunt mai mult decât suficiente.

7.8.5 Convertorul sigma-delta

Ultima familie inversează complet compromisul obișnuit: în loc să facă puține măsurători foarte fine, face foarte multe măsurători foarte grosolane și obține finețea prin mediere. Un convertor *sigma-delta* conține un modulator alcătuit dintr-un sumator, un integrator, un comparator de un singur bit și un convertor numeric-analogic tot de un bit, legat în reacție. Sumatorul scade din tensiunea de intrare ieșirea convertorului de reacție, diferența este integrată, iar comparatorul decide semnul rezultatului integrat, producând un 1 sau un 0. Această decizie comandă convertorul de reacție, care aplică la sumator fie V_{REF+} , fie V_{REF-} , închizând bucla.

Bucloa se comportă ca un sistem care își caută echilibrul: pentru ca integratorul să nu diveargă, densitatea de unități din fluxul de ieșire trebuie să fie proporțională cu tensiunea de intrare. Un semnal aflat la mijlocul domeniului produce un flux cu tot atâția de 1 cât de 0; unul aflat la trei sferturi produce trei de 1 la fiecare 0. Informația nu se mai află, așadar, în amplitudinea unei mărimi, ci în densitatea unui șir de biți, iar recuperarea ei se face printr-un filtru numeric care numără unitățile pe o fereastră lungă și, prin aceasta, mediază. Modulatorul lucrează la o frecvență de zeci sau sute de ori mai mare decât banda utilă, de unde numele de *supraeșantionare*.

Rezultatul este o rezoluție greu de atins altfel, de 16, 24 sau chiar 32 de biți, cu un hardware analogic remarcabil de simplu și puțin sensibil la toleranțele componentelor. Prețurile plătite sunt viteza mică și latența filtrului numeric, care are nevoie de multe eșantioane înainte de primul rezultat valid. Convertoarele sigma-delta domină aplicațiile de măsurare lentă și de mare exactitate: cântare electronice, termometre de precizie, achiziție audio.

7.8.6 Compararea arhitecturilor

Tabelul 7.3 rezumă compromisul dintre cele patru familii. Citit pe verticală, el arată că nu există o arhitectură superioară în absolut, ci numai o arhitectură potrivită pentru un anumit set de cerințe.

Tabela 7.3 Compararea arhitecturilor de convertoare analog-numerice

Arhitectură	Viteză	Rezoluție	Utilizare tipică
Paralel (flash)	foarte mare	mică (până la 8 biți)	Osciloscoape, achiziție video, radio numeric
Aproximări succesive	medie-mare	medie-mare (10-16 biți)	Microcontrolere, achiziție de date de uz general
Rampă simplă sau dublă	mică	mare	Multimetre, măsurători de laborator
Sigma-delta	mică	foarte mare (16-32 biți)	Cântare, termometre de precizie, audio

7.9 Erori de conversie

Pe lângă eroarea de cuantizare, care este de principiu, un convertor real prezintă abateri provenite din imperfecțiunile componentelor. Ele se descriu comparând caracteristica reală de transfer cu dreapta ideală și se măsoară, prin convenție, în multipli ai pasului de cuantizare.

Eroarea de offset este deplasarea pe verticală a întregii caracteristici: pentru o tensiune de intrare nulă, ieșirea nu este codul zero, ci un cod diferit de zero, sau invers, codul zero se menține și pentru tensiuni sensibil pozitive. Ea provine din tensiunea de decalaj a comparatoarelor și a amplificatoarelor de intrare, din curenții de scurgere și din variațiile de temperatură, și se măsoară aplicând la intrare masa analogică: orice abatere de la zero este chiar eroarea de offset. Fiind o deplasare constantă, ea se compensează ușor, prin scăderea unei constante din toate rezultatele.

Eroarea de amplificare, numită și eroare de câștig, este modificarea pantei caracteristicii: chiar dacă originea este corectă, codul obținut la capătul superior al domeniului diferă de cel ideal. Cauza principală este inexactitatea tensiunii de referință, la care se adaugă toleranțele componentelor din rețeaua internă. Ea se determină prin metoda punctelor de capăt, aplicând succesiv o tensiune cunoscută la fiecare limită a domeniului, și se compensează, fiind multiplicativă, printr-un factor de corecție determinat la calibrare.

Neliniaritatea diferențială, prescurtată DNL, măsoară cât de neegale sunt treptele caracteristicii. Pentru codul de rang n , cu lățimea reală W_n , ea se definește ca raportul $DNL(n) = (W_n - 1 \text{ LSB})/1 \text{ LSB}$. O valoare nulă înseamnă o treaptă ideală; o valoare de +0,125 înseamnă o treaptă cu 12,5 % mai lată decât ar trebui. Cazul cel mai grav este $DNL = -1$, care corespunde unei trepte de lățime nulă, adică unui cod care nu apare nicio dată la ieșire, oricare ar fi tensiunea de intrare.

Neliniaritatea integrală, prescurtată INL, este abaterea maximă a caracteristicii reale de la dreapta ideală și se obține prin însumarea neliniarităților diferențiale ale codurilor precedente, $INL(n) = \sum_{k=1}^{n-1} DNL(k)$. Dacă, de exemplu, primele două coduri au $DNL(1) = 0,125$

și $DNL(2) = 0,25$, atunci $INL(3) = 0,375$ LSB. Spre deosebire de offset și de câștig, neliniaritatea nu poate fi corectată printr-o operație simplă, ci numai printr-un tabel de corecție obținut prin calibrare punct cu punct, motiv pentru care ea limitează în final exactitatea unui convertor. Pentru ADC-ul din ATmega328P, foaia de catalog indică o exactitate absolută de ordinul a doi pași de cuantizare, deci din cei zece biți nominali aproximativ nouă sunt de încredere.

7.10 Modulul ADC al microcontrolerului ATmega328P

Convertorul integrat în ATmega328P este de tip cu aproximări succesive, are rezoluția de zece biți și este precedat de un multiplexor analogic care îi permite să deservească, pe rând, mai multe intrări: șase la capsula DIP folosită pe Arduino Uno, notate ADC0-ADC5 și accesibile la pinii marcați A0-A5. Blocul de conversie este unul singur, deci canalele nu pot fi citite simultan, ci numai succesiv, cu comutarea multiplexorului între conversii.

7.10.1 Registrul ADMUX

Registrul ADMUX concentrează cele trei decizii care privesc *ce se măsoară și față de ce*. Biții REFS1 și REFS0 aleg sursa de referință, conform Tabelului 7.1. Biții MUX3:0 selectează canalul de intrare, valorile de la 0000 la 0101 corespunzând canalelor ADC0-ADC5, iar câteva combinații speciale dând acces la senzorul intern de temperatură, la o tensiune fixă de 1,1 V și la masă. Bitul ADLAR decide alinierea rezultatului: pe zero, rezultatul este aliniat la dreapta, cu cei opt biți inferiori în ADCL și cei doi superiori în ADCH; pe unu, este aliniat la stânga, ceea ce permite citirea celor mai semnificativi opt biți dintr-un singur registru, ADCH.

7.10.2 Registrul ADCSRA

Registrul ADCSRA conține comenzile și starea modulului. Bitul ADEN alimentează blocul analogic; el trebuie activat înaintea oricărei conversii și poate fi dezactivat pentru a economisi curent atunci când convertorul nu este folosit. Bitul ADSC declanșează o conversie atunci când este scris cu 1 și rămâne 1 pe toată durata acesteia, revenind singur la zero la terminare; interogarea lui este cea mai simplă metodă de a aștepta rezultatul. Bitul ADATE validează declanșarea automată dintr-o sursă selectată prin biții ADTS2:0 din registrul ADCSRB: o depășire de timer, o comparare, o întrerupere externă sau chiar terminarea conversiei precedente, caz în care convertorul lucrează continuu, în regim numit *free running*. Bitul ADIF este indicatorul de terminare a conversiei, iar ADIE validează întreruperea asociată, cu vectorul ADC_vect. În sfârșit, biții ADPS2:0 fixează factorul de prescalare al ceasului de conversie, conform Tabelului 7.2.

7.10.3 Registrele de rezultat și ordinea de citire

Rezultatul de zece biți nu încapă într-un registru de opt biți al unei mașini de opt biți, deci este împărțit în două: ADCL conține octetul inferior, ADCH pe cel superior. Citirea lor nu poate fi făcută printr-o singură instrucțiune, iar de aici apare o problemă de coerență identică cu cea întâlnită la registrele de 16 biți ale timerelor: dacă între cele două citiri

se termină o nouă conversie, cei doi octeți ar proveni din măsurători diferite, iar valoarea recompusă ar fi lipsită de sens.

Hardware-ul rezolvă problema printr-un mecanism de blocare: în momentul în care programul citește ADCL, accesul convertorului la perechea de registre este blocat, iar blocarea se ridică abia la citirea lui ADCH. Consecința este o regulă strictă de ordine: ADCL se citește *primul*, iar ADCH imediat după. Dacă programul citește numai ADCH, sau dacă citește ADCL și uită să citească ADCH, registrele rămân blocate și conversiile ulterioare se pierd, fără nicio eroare.

Observație

Pe ATmega328P, ADCL trebuie citit înaintea lui ADCH. Citirea lui ADCL blochează registrele de rezultat, iar citirea lui ADCH le deblochează. În C se poate folosi și pseudoregistrul de 16 biți ADCW (sau ADC), pentru care compilatorul generează automat cele două citiri în ordinea corectă. Excepția este cazul ADLAR = 1 cu rezoluție de opt biți, când se citește numai ADCH.

7.10.4 Citirea cu funcțiile de bibliotecă

Cel mai simplu mod de a obține o măsurătoare este funcția `analogRead()`, care ascunde toată configurarea de mai sus. Ea selectează canalul, declanșează conversia, așteaptă terminarea ei prin interogarea bitului ADSC și returnează rezultatul pe zece biți, ca număr întreg între 0 și 1023.

Exemplu 7.1 Citirea unei intrari analogice cu functiile de biblioteca

```

1  const uint8_t PIN_SENZOR = A0;
2
3  void setup() {
4      Serial.begin(9600);
5      // analogReference(DEFAULT); // AVCC, valoarea implicita
6      // analogReference(INTERNAL); // referinta interna de 1,1 V
7  }
8
9  void loop() {
10     int brut = analogRead(PIN_SENZOR); // 0 ... 1023, dureaza cca
        ↪ 104 us
11     Serial.println(brut);
12     delay(200);
13 }
```

Prima citire de după o schimbare a referinței prin `analogReference()` poate fi greșită, pentru că sursa de referință are nevoie de timp pentru a se stabili pe condensatorul de la AREF. Practica sănătoasă este o citire de probă, al cărei rezultat se aruncă, după orice schimbare de referință sau de canal.

7.10.5 Aceeași citire scrisă direct pe registre

Varianta cu acces direct la registre arată explicit ce se întâmplă și permite lucruri pe care funcția de bibliotecă nu le oferă: schimbarea factorului de prescalare, declanșarea automată dintr-un timer, tratarea prin întrerupere.

Exemplu 7.2 Configurarea și citirea ADC-ului direct pe registre

```

1  #include <avr/io.h>
2
3  void adc_init(void) {
4      // referinta AVCC, rezultat aliniat la dreapta, canal 0 selectat
      ↪ provizoriu
5      ADMUX = (1 << REFS0);
6      // activeaza modulul si alege prescalarea 128: 16 MHz / 128 =
      ↪ 125 kHz
7      ADCSRA = (1 << ADEN) | (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0
      ↪ );
8      // dezactiveaza tamponul digital pe ADC0, pentru mai putin
      ↪ zgomot si consum
9      DIDR0 = (1 << ADC0D);
10 }
11
12 uint16_t adc_citeste(uint8_t canal) {
13     // pastreaza bitii de referinta, inlocuieste doar selectia
      ↪ canalului
14     ADMUX = (ADMUX & 0xF0) | (canal & 0x0F);
15
16     _delay_us(10);                // stabilizarea intrarii dupa
      ↪ comutare
17
18     ADCSRA |= (1 << ADSC);        // porneste conversia
19     while (ADCSRA & (1 << ADSC)) {
20         ;                        // asteapta: ADSC revine la 0
      ↪ la terminare
21     }
22
23     uint8_t inferior = ADCL;      // ADCL SE CITESTE PRIMUL:
      ↪ blocheaza registrele
24     uint8_t superior = ADCH;     // citirea lui ADCH
      ↪ deblocheaza registrele
25     return ((uint16_t)superior << 8) | inferior;
26 }

```

Ordinea operațiilor din `adc_init()` nu este întâmplătoare: factorul de prescalare se scrie în același timp cu bitul de activare, pentru ca prima conversie să se desfășoare deja cu ceasul corect. Masca `ADMUX & 0xF0` din funcția de citire păstrează biții de referință și de aliniere, modificând numai selecția canalului; o scriere neatentă a întregului registru ar reveni implicit la referința de pe AREF și ar falsifica toate măsurătorile.

Aceeași citire se poate face și prin întrerupere, ceea ce eliberează procesorul pe durata celor 104 μ s ale conversiei:

Exemplu 7.3 Conversie declansata software, cu tratare prin intrerupere

```

1  volatile uint16_t rezultat = 0;
2  volatile uint8_t  gata      = 0;
3
4  void adc_start(void) {
5      ADCSRA |= (1 << ADIE);           // valideaza intreruperea de
        ↪ terminare
6      sei();
7      ADCSRA |= (1 << ADSC);           // porneste conversia si
        ↪ revine imediat
8  }
9
10 ISR(ADC_vect) {
11     rezultat = ADCW;                  // citire pe 16 biti, in
        ↪ ordinea corecta
12     gata = 1;                         // programul principal va
        ↪ prelucra valoarea
13 }
```

7.10.6 Transformarea valorii brute în tensiune

Valoarea returnată de convertor este un număr fără unitate de măsură; pentru a deveni o tensiune, ea trebuie înmulțită cu pasul de cuantizare, adică raportată la referință. Calculul se poate face în virgulă mobilă, care este comod și clar, sau în aritmetică întreagă, care este mult mai rapidă pe un microcontroler fără coprocesor matematic.

Exemplu 7.4 Conversia valorii brute in tensiune, in virgula mobila si in intregi

```

1  const float V_REF = 5.0f;           // atentie: valoarea reala a
        ↪ alimentarii
2
3  // varianta in virgula mobila: clara, dar lenta si consumatoare de
        ↪ memorie
4  float adc_in_volti(uint16_t brut) {
5      return (brut * V_REF) / 1024.0f;
```

```

6  }
7
8  // varianta in aritmetica intreaga: rezultatul in milivolti
9  // 5000 mV * 1023 = 5115000, incape in uint32_t
10 uint16_t adc_in_milivolti(uint16_t brut) {
11     return (uint16_t)((uint32_t)brut * 5000UL) / 1024UL;
12 }
13
14 // exemplu de utilizare pentru un divizor rezistiv 1:2 (masurarea
    ↪ unei baterii)
15 uint16_t tensiune_baterie_mv(uint16_t brut) {
16     return adc_in_milivolti(brut) * 2;
17 }

```

Împărțirea la 1024 merită o observație: fiind o putere a lui doi, compilatorul o înlocuiește cu o deplasare la dreapta cu zece poziții, operație de câțiva cicluri mașină. Dacă s-ar scrie în locul ei o împărțire la 1023, ca în formula cu $2^n - 1$, compilatorul ar genera o rutină de împărțire completă, de zeci de ori mai lentă, pentru un câștig de exactitate inferior unei miimi.

7.11 Greșeli tipice și recomandări practice

Experiența laboratorului arată că erorile de măsurare analogică se repetă, în proporție covârșitoare, sub aceleași câteva forme. Merită trecute în revistă, pentru că fiecare dintre ele are o explicație care ține de materia acestui capitol.

Impedanța prea mare a sursei. Este cauza cea mai frecventă a rezultatelor care „nu se mișcă” sau care depind de canalul citit anterior. Un divizor construit din rezistoare de sute de kilohmi, un termistor cu rezistență mare la temperatură joasă sau un electrod de măsurare nu pot încărca în timp util condensatorul de 14 pF al circuitului de eșantionare, iar rezultatul este sistematic mai apropiat de valoarea măsurată anterior decât de cea reală. Simptomul caracteristic este tocmai această „memorie” a canalului precedent, care apare pentru că, la comutarea multiplexorului, condensatorul păstrează sarcina de la conversia trecută. Remediile sunt trei: coborârea impedanței sursei sub 10 kΩ, intercalarea unui amplificator repetor sau, ca soluție de compromis, citirea de două ori a aceluiași canal și folosirea celei de a doua valori.

Lipsa filtrului anti-alias. Un montaj care citește un semnal fără să îi limiteze în prealabil banda va aduna, în rezultat, toate perturbațiile de frecvență înaltă din mediu: comutările sursei în comutație, semnalul PWM al unui motor din apropiere, armonicile rețelei. Aceste componente nu apar ca zgomot recunoscutibil, ci ca variații lente și verosimile ale mărimii măsurate, produse de aliasing, ceea ce le face deosebit de înșelătoare. Un simplu grup RC cu frecvența de tăiere de câteva sute de hertzi, plasat cât mai aproape de pinul de intrare, rezolvă majoritatea situațiilor.

Referința lăsată pe AVCC atunci când alimentarea nu este de 5 V. O placă alimentată prin USB, un montaj alimentat din baterie sau unul care comandă o sarcină de curent mare vor avea o alimentare diferită de valoarea nominală și variabilă în timp. Programul care presupune 5,000 V va produce rezultate false proporțional. Soluțiile sunt: fie măsurarea ratiometrică, dacă senzorul este alimentat din aceeași tensiune; fie trecerea la referința internă de 1,1 V pentru semnale mici; fie măsurarea efectivă a alimentării, folosind faptul că referința internă poate fi citită ca și cum ar fi un canal de intrare, ceea ce permite deducerea valorii lui AVCC prin calcul.

Media aritmetică folosită ca panaceu. Obiceiul de a citi de o sută de ori și de a face media este uneori util, dar este adesea aplicat ca un ritual, fără a înțelege ce poate și ce nu poate el să facă. Media reduce zgomotul *aleator*, necorelat de la un eșantion la altul, în raport cu rădăcina pătrată a numărului de eșantioane: o sută de citiri reduc zgomotul de zece ori, adică aduc puțin peste trei biți suplimentari de rezoluție utilă, cu condiția ca semnalul măsurat să conțină deja o cantitate de zgomot comparabilă cu un pas de cuantizare. Media nu corectează în schimb absolut nimic din erorile *sistematice*: un offset rămâne offset, o referință greșită rămâne greșită, iar o componentă intrusă prin aliasing rămâne acolo unde este, pentru că nu este zgomot aleator, ci un semnal perfect determinist. Mai mult, medierea a o sută de citiri consumă 10,4 ms, ceea ce reduce corespunzător banda utilă a măsurătorii. Media este un filtru trece-jos numeric, iar ca orice filtru trece-jos cumpără liniște cu preț de întârziere.

La aceste patru se adaugă câteva reguli de montaj care nu costă nimic și previn multe ore de căutare a unei erori inexistente în software: legarea masei analogice la masa numerică într-un singur punct, apropierea condensatoarelor de decuplare de pinii de alimentare, dezactivarea tamponului digital al pinilor folosiți ca intrări analogice prin registrul DIDRO și evitarea comutării unor sarcini puternice chiar în intervalul în care se face o conversie.

7.12 Concluzii

Convertorul analog-numeric este puntea dintre lumea continuă a mărimilor fizice și lumea discretă a memoriei unui microcontroler. Traversarea acestei punți se face în trei etape obligatorii - eșantionarea, care discretizează timpul; cuantizarea, care discretizează amplitudinea; și codificarea cu stocare, care face rezultatul accesibil programului - încadrate de un filtru anti-alias înaintea lor și de un circuit de eșantionare și reținere care imobilizează valoarea pe durata conversiei.

Fiecare dintre aceste etape impune o condiție cantitativă pe care proiectantul nu o poate ocoli. Eșantionarea cere ca frecvența de prelevare să depășească dublul celei mai înalte frecvențe din semnal, altfel apare aliasing, iar componentele rapide se deghizează definitiv în componente lente. Cuantizarea introduce o eroare de cel mult un pas $q = V_{REF}/2^r$, ceea ce înseamnă aproximativ 4,9 mV pentru zece biți la 5 V, iar rezoluția trebuie aleasă în raport cu variația utilă a mărimii măsurate, nu maximizată în abstract. Timpul de conversie fixează plafonul ratei de eșantionare; pe ATmega328P, cele 13 cicluri ale ceasului de 125 kHz obținut prin prescalare cu 128 dau 104 μ s pe conversie, deci aproximativ 9600 de eșantioane

pe secundă și o bandă utilă de câteva sute de hertzi până la un kilohertz.

Tensiunea de referință este mărimea de care depinde totul și de care se ține cel mai puțin seama. Ea fixează simultan domeniul de măsurare și rezoluția efectivă, iar zgomotul sau imprecizia ei se transmit, nediminuate, în toate rezultatele; alegerea între AVCC, referința internă de 1,1 V și o sursă externă nu este o opțiune de configurare, ci o decizie de proiectare.

În sfârșit, arhitecturile disponibile nu se ierarhizează, ci se aleg. Convertorul paralel cumpără viteza cu un număr exponențial de comparatoare; cel cu rampă cumpără exactitatea cu timp; cel sigma-delta cumpără rezoluția cu supraeșantionare și latență; iar cel cu aproximări succesive, o căutare binară pe tensiune în care fiecare comparație elimină jumătate din posibilitățile rămase, oferă compromisul echilibrat care l-a făcut arhitectura standard a microcontrolerelor.

Întrebări de verificare

1. Explicați de ce o mărime fizică nu poate fi memorată exact într-un sistem numeric și în ce sens conversia analog-numerică reprezintă o pierdere deliberată de informație. Ce mărime din lanțul de conversie controlează dimensiunea acestei pierderi?
2. Descrieți rolul circuitului de eșantionare și reținere. Ce s-ar întâmpla, concret, la un convertor cu aproximări succesive de zece biți, dacă tensiunea de intrare s-ar modifica cu jumătate din domeniu între prima și a doua comparație?
3. Un senzor are o rezistență internă de $470\text{ k}\Omega$ și este legat direct la o intrare analogică a unui ATmega328P. Calculați constanta de timp a încărcării condensatorului de eșantionare de 14 pF și comparați timpul necesar încărcării complete cu durata fazei de achiziție. Ce soluție propuneți?
4. Enunțați teorema eșantionării și explicați intuitiv de ce apare factorul doi. De ce inegalitatea trebuie să fie strictă și ce se întâmplă la exact două eșantioane pe perioadă?
5. Un semnal de 6 kHz este eșantionat la 8 kHz . Determinați frecvența aparentă a semnalului reconstituit și arătați, prin calculul câtorva eșantioane, de ce cele două semnale sunt indistinctibile. De ce fenomenul nu poate fi corectat prin prelucrare numerică ulterioară?
6. De ce filtrul anti-alias trebuie să fie analogic și nu numeric? Ce constrângere apare la alegerea rezistorului dintr-un filtru RC plasat la intrarea unui ADC și de ce?
7. Calculați pasul de cuantizare și eroarea maximă de cuantizare pentru un convertor de 10 biți cu referința de 5 V , respectiv cu referința internă de $1,1\text{ V}$. Un semnal care variază între 0 și $0,8\text{ V}$: pe ce referință se măsoară mai bine și cu ce câștig de rezoluție efectivă?
8. Explicați ce este o măsurătoare ratiometrică și de ce, în cazul unui potențiomtru alimentat din V_{CC} , referința pe AVCC dă rezultate corecte chiar dacă alimentarea nu

este de exact 5 V. Dați un exemplu de măsurătoare pentru care acest argument nu mai este valabil.

9. Un ATmega328P funcționează la 16 MHz. Calculați, pentru fiecare factor de prescalare disponibil, frecvența ceasului ADC și timpul unei conversii de 13 cicluri. Arătați care variante respectă domeniul recomandat de 50-200 kHz și ce se pierde alegând una din afara lui.
10. Descrieți pas cu pas conversia tensiunii de 1,8 V de către un convertor cu aproximări succesive de 4 biți cu referința de 5 V. Indicați, la fiecare pas, tensiunea de probă, decizia comparatorului și intervalul de incertitudine rămas, apoi calculați eroarea finală în pași de cuantizare.
11. Comparați convertorul paralel cu cel cu aproximări succesive din punctul de vedere al numărului de comparatoare, al timpului de conversie și al rezoluției realizabile. De ce un convertor paralel de 12 biți este practic nerealizabil în tehnologia obișnuită?
12. De ce registrul ADCL trebuie citit înaintea lui ADCH pe ATmega328P? Ce se întâmplă dacă un program citește numai ADCH și de ce eroarea este greu de depistat?
13. Un program citește de o sută de ori aceeași intrare analogică și face media rezultatelor. Explicați ce categorii de erori sunt reduse prin această operație și care rămân neschimbate. Cu cât scade zgomotul aleator și cu cât se reduce banda utilă a măsurătorii?

8. Comunicația serială asincronă

Sistemele de comunicație
Tipuri de comunicație
Comunicația serială sincronă și asincronă
Modulul UART
Formatul cadrului
Sincronizarea între emițător și receptor
Calculul ratei de transfer
Registrele modulului USART al microcontrolerului AT-mega328P
Aplicație: transmisia și recepția unui caracter
Recepția cu întrerupere și tamponul circular
De la UART la USART
Standardele RS-232 și RS-485
Concluzii
Întrebări de verificare

8.1 Sistemele de comunicație

Un microcontroler care lucrează singur este o excepție. În practică, aproape orice sistem încorporat este format din mai multe circuite care trebuie să schimbe informație între ele: un senzor de temperatură care raportează o valoare, un afișaj care trebuie să primească un text, un modul de memorie care păstrează un istoric, un calculator care primește datele măsurate pentru a le prelucra ulterior. Toate aceste schimburi presupun un mecanism prin care o informație existentă într-un registru al unui circuit ajunge, nealterată, într-un registru al altui circuit. Acest mecanism, luat împreună cu regulile care îl guvernează, formează un *sistem de comunicație*.

Nevoia de comunicație apare din câteva situații care se repetă în orice aplicație. Prima este interfațarea cu perifericele externe: senzori, actuatori, module de stocare, afișaje. Un accelerometru modern nu mai livrează o tensiune analogică proporțională cu accelerația, ci un număr pe 16 biți, pe care îl transmite serial la cerere; un afișaj grafic nu se comandă punct cu punct prin pini, ci prin comenzi trimise pe o interfață serială. A doua situație este comunicația între mai multe microcontrolere, atunci când sarcina se distribuie între unități specializate: un microcontroler se ocupă de bucla rapidă de reglaj a unui motor, altul de

interfața cu operatorul, iar cele două trebuie să se pună de acord. A treia este legătura cu dispozitivele destinate utilizatorului - calculator, telefon, tabletă - fie pentru configurare, fie pentru afișarea datelor, fie pentru depanarea programului. A patra, tot mai frecventă, este integrarea în rețele și în aplicații de tip *Internet of Things*, unde microcontrolerul trimite măsurători către un server sau primește comenzi de la o aplicație aflată la distanță.

Toate aceste cazuri au în comun o dificultate esențială: cele două capete ale legăturii sunt circuite *independente*, fiecare cu ceasul lui, cu alimentarea lui, cu programul lui. Nu există o memorie comună pe care să o citească amândouă. Informația trebuie deci codificată ca o variație de tensiune pe un fir, transmisă în timp, apoi decodificată la celălalt capăt. Întreaga materie a acestui capitol și a următorului constă în a răspunde la două întrebări: cum se codifică biții în timp și cum știe receptorul unde începe și unde se termină fiecare bit.

8.2 Tipuri de comunicație

Protocoalele de comunicație folosite în sistemele încorporate se pot clasifica după mai multe criterii, iar criteriile nu se exclud: același protocol este, în același timp, serial, asincron, duplex integral și punct la punct. Merită parcurse pe rând, pentru că fiecare dintre ele răspunde la o întrebare de proiectare diferită.

8.2.1 Serial față de paralel

Primul criteriu se referă la numărul de fire pe care circulă simultan biții unui cuvânt. În *comunicația paralelă*, cei opt biți ai unui octet sunt transmiși în același moment, pe opt fire distincte, plus un fir de validare care spune receptorului că datele de pe magistrală sunt stabile. În *comunicația serială*, cei opt biți sunt transmiși unul după altul, pe un singur fir, în opt intervale de timp succesive.

La prima vedere concluzia pare evidentă: legătura paralelă este de opt ori mai rapidă, deci ar trebui preferată. Realitatea a mers exact în sens invers, iar motivele merită înțelese, pentru că explică de ce practic toate interfețele moderne, de la USB și SATA până la interfața senzorilor de câțiva cenți, sunt seriale.

Primul motiv este numărul de pini. Un ATmega328P are 23 de pini de intrare-ieșire în total. O magistrală paralelă de 8 biți cu semnale de control ar consuma aproximativ jumătate dintre ei pentru un singur periferic. O interfață serială consumă doi. Pini sunt resursa cea mai scumpă a unui microcontroler, pentru că ei determină dimensiunea capsulei, iar capsula determină prețul și suprafața ocupată pe cablaj.

Al doilea motiv, mai subtil, este *decalajul de propagare* (în engleză *skew*). Cele opt fire ale unei magistrale paralele nu au niciodată exact aceeași lungime electrică: traseele de pe cablaj ocolesc diferit, capacitățile parazite diferă, iar sarcinile conectate diferă. Un bit ajunge cu câteva sute de picosecunde înaintea altuia. Cât timp durată unui bit este de ordinul microsecundelor, decalajul nu contează. Dacă însă se dorește creșterea frecvenței, durată unui bit scade, iar la un moment dat decalajul dintre fire devine comparabil cu durată bitului: receptorul citește, în aceeași fereastră de validare, bitul n de pe un fir și bitul $n+1$ de

pe altul. Frecvența unei magistrale paralele este deci limitată nu de electronica de capăt, ci de diferențele dintre fire, iar această limită se înrăutățește cu lungimea cablului. O legătură serială nu are această problemă: neexistând decât un fir de date, nu există nimic față de care acesta să se decaleze.

Al treilea motiv este diafonia. Opt fire alăturate care comută simultan se influențează reciproc prin cuplaj capacitiv și inductiv, iar zgomotul injectat crește cu viteza de comutare. Un singur fir, mai ales dacă este ecranat sau torsadat cu masa lui, se comportă mult mai bine. În fine, cablurile și conectoarele seriale sunt mai ieftine, mai subțiri și mai flexibile.

Rezultatul practic este că o legătură serială bine făcută poate funcționa la frecvențe cu două-trei ordine de mărime mai mari decât una paralelă de aceeași lungime, iar acest câștig de frecvență compensează cu mult pierderea celor opt fire simultane. Comunicația paralelă a rămas doar în interiorul circuitelor integrate și pe distanțe de câțiva centimetri, acolo unde decalajul este neglijabil, de exemplu la interfața cu o memorie externă.

8.2.2 Sincron față de asincron

Al doilea criteriu privește modul în care receptorul află *când* să citească linia de date. Într-o comunicație *sincronă*, emițătorul transmite, pe un fir separat, un semnal de ceas care marchează fiecare bit; receptorul nu are decât să eșantioneze linia de date pe frontul indicat de acest ceas. Sincronizarea este atunci perfectă prin construcție, indiferent de viteză și indiferent de precizia oscilatoarelor celor două capete. Protocoalele SPI și I²C funcționează astfel.

Într-o comunicație *asincronă* nu există fir de ceas. Receptorul trebuie să deducă singur momentele de eșantionare, folosind propriul oscilator, pornind de la un reper transmis chiar în fluxul de date. Acest reper este tranziția care marchează începutul unui cadru. Costul este că cele două capete trebuie să fi convenit *dinainte* asupra vitezei de transmisie, iar oscilatoarele lor trebuie să fie suficient de apropiate ca frecvență pentru ca abaterea acumulată pe durata unui cadru să nu ducă la eșantionarea unui bit vecin. Câștigul este un fir mai puțin și, mai ales, posibilitatea ca cele două dispozitive să fie complet independente, fără nicio legătură de tact între ele.

8.2.3 Simplex, semiduplex, duplex integral

Al treilea criteriu privește sensul în care poate circula informația. O legătură *simplex* transmite într-un singur sens, permanent: un emițător radio către un receptor, un senzor care raportează continuu fără să primească vreodată comenzi. O legătură *semiduplex* (half-duplex) poate transmite în ambele sensuri, dar nu simultan; cele două capete folosesc pe rând același mediu, iar protocolul trebuie să prevadă cine are dreptul să vorbească la un moment dat. O legătură *duplex integral* (full-duplex) transmite simultan în ambele sensuri, ceea ce presupune fie două fire de date, fie două canale separate în frecvență. UART-ul cu două fire, RxD și TxD, este duplex integral: un microcontroler poate primi un caracter în timp ce transmite altul, pentru că recepția și emisia folosesc circuite și fire distincte.

8.2.4 Punct la punct față de magistrală

Ultimul criteriu privește numărul de participanți. O legătură *punct la punct* are exact două capete și nu are nevoie de adresare: tot ce se transmite pe fir este destinat celuilalt. O *magistrală* (topologie multipunct) leagă mai multe dispozitive la aceleași fire, ceea ce impune două mecanisme suplimentare: un mecanism de adresare, prin care se precizează cui îi este destinat mesajul, și un mecanism de arbitrare, care să împiedice două dispozitive să emită simultan. UART-ul este, în forma sa de bază, punct la punct; standardul RS-485, discutat la sfârșitul capitolului, îl extinde la o magistrală cu până la 32 de dispozitive.

La acestea se adaugă clasificarea după mediul de transmisie, între comunicație prin fir - protocoalele seriale clasice, Ethernet - și comunicație fără fir - Bluetooth, Wi-Fi, ZigBee. Este important de observat că un modul radio de tip Bluetooth se prezintă, față de microcontroler, tot ca o interfață serială asincronă: microcontrolerul îi trimite octeți prin UART, iar modulul se ocupă de partea de radiofrecvență. Din acest motiv, stăpânirea comunicației seriale asincrone deschide accesul la o categorie foarte largă de periferice.

8.3 Comunicația serială sincronă și asincronă

Am văzut că, în absența unui fir de ceas, receptorul trebuie să genereze singur momentele de eșantionare. Merită insistat asupra a ceea ce înseamnă concret acest lucru, pentru că aici se află sursa majorității problemelor practice.

Să presupunem că emițătorul transmite un octet ca opt niveluri succesive de tensiune, fiecare durând exact T_b secunde, și că linia rămâne în starea inactivă înainte și după. Receptorul vede o singură tensiune care variază în timp. Ca să extragă cei opt biți, el trebuie să știe două lucruri: *unde* începe primul bit și *cât* durează un bit. A doua informație nu se poate obține din semnal, pentru că, dacă opt biți consecutivi au aceeași valoare, linia nu are nicio tranziție din care să se deducă ritmul. Ea trebuie deci stabilită prin convenție: ambele capete sunt configurate, înainte de a începe schimbul, cu aceeași *rată de transfer*. Prima informație, în schimb, poate fi transmisă: este suficient ca fiecare cadru să înceapă cu o tranziție recunoscută, pe care receptorul o folosește drept origine a timpului.

Aceasta este întreaga idee a comunicației seriale asincrone. Sincronizarea nu este continuă, ca la protocoalele sincrone, ci se reface la începutul fiecărui cadru, iar între două resincronizări receptorul lucrează pe încredere, măsurând timpul cu propriul oscilator. Cu cât cadrul este mai lung, cu atât mai mult are de mers receptorul pe încredere și cu atât mai mică trebuie să fie diferența dintre cele două ceasuri.

8.4 Modulul UART

Blocul hardware care implementează această schemă se numește *UART*, prescurtare de la *Universal Asynchronous Receiver Transmitter*. El este, în esență, un traducător bidirecțional între lumea paralelă din interiorul microcontrolerului, unde un octet există simultan pe opt linii ale magistralei interne, și lumea serială de pe fir, unde același octet există ca o succesiune de niveluri în timp.

Modulul folosește două linii de semnal. Linia **TxD** (*transmit data*) este ieșirea pe care microcontrolerul emite; linia **RxD** (*receive data*) este intrarea pe care primește. La conectarea a două dispozitive, cele două linii se încrucișează: **TxD** al primului merge la **RxD** al celui de al doilea și invers. Este greșeala clasică a începătorului să lege **TxD** la **TxD**, caz în care nu se transmite nimic și nici nu se arde nimic, ceea ce face eroarea greu de depistat. La acestea se adaugă obligatoriu o legătură de masă comună, pentru că nivelul logic este definit ca o tensiune *față de masă*; fără masă comună, cele două capete nu au aceeași referință și comunicația nu funcționează sau funcționează imprevizibil. Pe ATmega328P, cele două semnale sunt disponibile pe pinii PD0 (**RxD**) și PD1 (**TxD**), care corespund pinilor digitali 0 și 1 ai plăcii Arduino Uno.

Comportarea modulului este determinată de patru parametri, care trebuie să fie identici la ambele capete:

- **Numărul de biți de date**, în general între 5 și 9, cel mai adesea 8. Bitul cel mai puțin semnificativ se transmite primul.
- **Bitul de paritate**, care poate lipsi sau poate fi de tip par ori impar. La paritatea pară, bitul suplimentar se alege astfel încât numărul total de biți egali cu 1 din cadru, incluzând bitul de paritate, să fie par.
- **Rata de transfer**, exprimată în biți pe secundă. Valorile uzuale sunt 9600, 19200, 38400, 57600 și 115200.
- **Numărul de biți de stop**, unul sau doi.

Aceste configurații se notează prescurtat prin numărul de biți de date, litera care indică paritatea (**N** pentru absența ei, **E** pentru pară, **O** pentru impară) și numărul de biți de stop. Notăția **8N1**, adică opt biți de date, fără paritate, un bit de stop, este de departe cea mai folosită și este configurația implicită a aproape tuturor terminalelor seriale.

Observație

Termenii *baud* și *bit pe secundă* nu sunt sinonimi în general. Baudul măsoară numărul de simboluri transmise pe secundă, iar un simbol poate transporta mai mulți biți dacă modulația folosește mai mult de două niveluri. La UART, codificarea este binară, cu două niveluri, deci un simbol înseamnă exact un bit, iar cele două mărimi coincid numeric. Din acest motiv se poate spune, fără greșală, „9600 de baudzi” sau „9600 de biți pe secundă”.

8.5 Formatul cadrului

Codificarea folosită pe fir este de tip *NRZ* (*Non Return to Zero*): fiecare bit este reprezentat printr-un nivel constant menținut pe toată durata bitului, fără revenire la o valoare de repaus între biți. Un 1 logic este un nivel ridicat menținut T_b secunde, un 0 logic este un nivel coborât menținut tot T_b secunde. Consecința acestei alegeri este că un șir lung de biți identici nu produce nicio tranziție pe linie, ceea ce explică de ce receptorul nu poate recupera ceasul din semnal și de ce rata de transfer trebuie convenită dinainte.

În starea de repaus, când nu se transmite nimic, linia este menținută în 1 logic. Această alegere nu este arbitrară: ea permite distingerea liniei libere de linia întreruptă sau de

emițătorul oprit, care ar da un 0 permanent, și oferă o tranziție descrescătoare clară la începutul fiecărui cadru.

Cadrul propriu-zis are următoarea structură, în ordinea transmiterii:

- Un **bit de start**, întotdeauna 0 logic. Rolul lui este exclusiv de sincronizare: el produce, prin trecerea liniei din 1 în 0, frontul descrescător pe care receptorul îl folosește ca origine a timpului pentru întregul cadru. Nu transportă informație utilă.
- **Biții de date**, între 5 și 9, transmiși începând cu bitul cel mai puțin semnificativ.
- Eventual un **bit de paritate**, calculat asupra biților de date.
- Unul sau doi **biți de stop**, întotdeauna 1 logic. Rolul lor este de a garanta că linia revine în starea de repaus înainte de următorul cadru, astfel încât frontul descrescător al următorului bit de start să fie detectabil. Doi biți de stop se folosesc atunci când receptorul are nevoie de mai mult timp pentru a prelucra octetul primit.

Figura 8.1 arată forma de undă a unui cadru complet, pentru caracterul 'A', al cărui cod ASCII este 0x41, adică 01000001 în binar, cu paritate pară și un bit de stop. Se observă ordinea inversată a biților de date pe fir: primul transmis este $D_0 = 1$, ultimul este $D_7 = 0$.

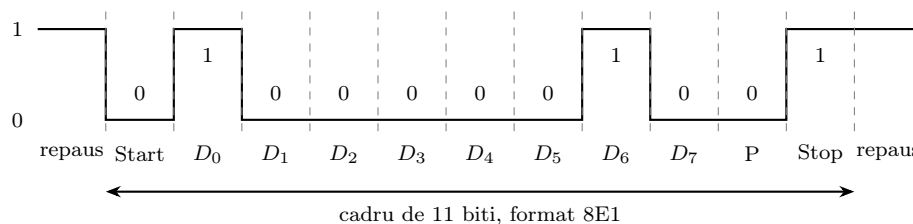


Figura 8.1 Formatul unui cadru UART pentru caracterul 'A' (0x41), cu opt biți de date, paritate pară și un bit de stop. Bitul cel mai puțin semnificativ se transmite primul.

O observație importantă privește randamentul acestei codificări. Pentru a transmite opt biți utili, formatul 8N1 pune pe fir zece biți, iar formatul 8E1 unsprezece. Din capacitatea brută a legăturii se pierde deci între 20 și 27 de procente pe sincronizare și pe control. La 9600 de baudzi în format 8N1, un cadru durează $10/9600 \approx 1,04$ ms, deci se pot transmite cel mult 960 de caractere pe secundă. La 115200 de baudzi, durata unui cadru scade la aproximativ $86,8 \mu s$, adică 11 520 de caractere pe secundă. Aceste cifre vor conta atunci când vom discuta despre tamponul de recepție.

Bitul de paritate merită și el o precizare. El permite detectarea unui număr impar de erori de bit, dar nu permite corectarea lor și nu detectează un număr par de erori. Într-o legătură scurtă, între două plăci de pe același cablaj, probabilitatea unei erori este atât de mică încât paritatea este de obicei dezactivată; ea devine utilă în medii perturbate electric sau pe cabluri lungi, unde însă un mecanism mai puternic, cum ar fi o sumă de control calculată pe un mesaj întreg, este oricum de preferat.

8.6 Sincronizarea între emițător și receptor

Am stabilit că, în comunicația asincronă, emițătorul și receptorul au ceasuri complet independente. Rămâne de văzut cum reușește totuși receptorul să eșantioneze fiecare bit

în locul potrivit și cât de mult pot diferi cele două ceasuri înainte ca lucrurile să nu mai funcționeze.

8.6.1 Eșantionarea la mijlocul bitului

Prima idee este simplă și decisivă: receptorul nu trebuie să eșantioneze linia la începutul unui bit, ci la *mijlocul* lui. Motivul este că marginile bitului sunt zonele cele mai nesigure. Acolo se produce tranziția, iar tranziția nu este instantanee: are un timp de creștere determinat de capacitatea liniei și de impedanța de ieșire a emițătorului, poate avea supracreșteri și oscilații, iar momentul exact în care se produce este afectat de jitterul emițătorului. La mijlocul bitului, în schimb, nivelul este stabil și cât se poate de departe de ambele tranziții vecine. Eșantionând acolo, receptorul își asigură cea mai mare marjă posibilă față de orice eroare de temporizare.

8.6.2 Supraeșantionarea de 16 ori

Pentru a găsi mijlocul bitului, receptorul nu lucrează cu un ceas de frecvența ratei de transfer, ci cu un ceas de 16 ori mai rapid. Blocul de recepție eșantionează linia RxD de $s = 16$ ori pe durata fiecărui bit, iar din poziția acestor eșantioane deduce unde se află mijlocul.

Mecanismul funcționează astfel. Cât timp linia este în repaus, receptorul așteaptă un front descrescător. În momentul în care îl detectează, presupune că a început bitul de start și pornește un numărător de eșantioane. Eșantioanele numărate 8, 9 și 10 cad, prin construcție, în jurul mijlocului bitului de start; receptorul le citește pe toate trei și decide prin vot majoritar. Dacă majoritatea indică 0 logic, frontul este confirmat ca bit de start valid și receptorul își fixează definitiv referința de timp; dacă majoritatea indică 1, frontul a fost un impuls parazit, iar receptorul revine în așteptare. Această verificare respinge automat impulsurile de zgomot mai scurte decât o jumătate de bit.

Odată referința fixată, receptorul nu mai are nevoie de nicio tranziție: el numără câte 16 eșantioane pentru fiecare bit următor și citește, de fiecare dată, eșantioanele 8, 9 și 10, decidând tot prin vot majoritar. Votul majoritar oferă imunitate la un impuls parazit scurt care ar afecta un singur eșantion. Biții astfel recuperați intră unul câte unul într-un registru de deplasare intern, iar la sfârșitul cadrului, după validarea bitului de stop, conținutul registrului de deplasare este transferat într-un registru tampon de unde programul îl poate citi. Figura 8.2 ilustrează acest proces.

Supraeșantionarea explică, în treacăt, și de ce receptorul are nevoie de un ceas de 16 ori mai rapid decât rata de transfer. La 115200 de baudzi, ceasul intern al blocului de recepție lucrează la $16 \times 115200 = 1,8432$ MHz, ceea ce înseamnă că oscilatorul microcontrolerului trebuie să poată furniza această frecvență prin divizare întreagă. Vom vedea imediat că tocmai aici apare principala limitare practică.

8.6.3 De ce eroarea de ceas se acumulează

Punctul esențial de înțeles este că receptorul se resincronizează o singură dată pe cadru, pe frontul bitului de start, și că toate momentele de eșantionare ulterioare sunt măsurate

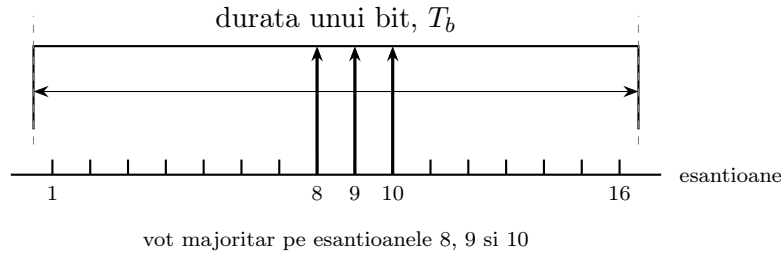


Figura 8.2 Supraeșantionarea de 16 ori pe bit la receptorul USART al microcontrolerelor AVR. Decizia asupra nivelului logic se ia prin vot majoritar pe eșantioanele 8, 9 și 10, situate în jurul mijlocului bitului.

față de acel front cu propriul lui ceas. Dacă ceasul receptorului are o eroare relativă e față de cel al emițătorului, atunci intervalul pe care receptorul îl consideră un bit este de fapt $T_b(1 + e)$. Eroarea nu se anulează de la un bit la altul, ci se *adună*: după n biți, momentul la care receptorul crede că se află la mijlocul bitului n este deplasat cu $n \cdot e \cdot T_b$ față de mijlocul real.

Consecința este că bitul cel mai vulnerabil dintr-un cadru nu este primul, ci ultimul. Într-un cadru 8N1, mijlocul bitului de stop se află la 9,5 durate de bit după frontul de start; într-un cadru 8E1, la 10,5. Ca eșantionul să cadă încă în interiorul bitului, deplasarea acumulată trebuie să rămână sub o jumătate de bit, deci, în cazul 8N1,

$$9,5 \cdot |e| < 0,5 \quad \Rightarrow \quad |e| < 5,3\%.$$

Aceasta este limita geometrică ideală. În practică limita este mai strânsă din două motive. Primul este că receptorul nu detectează frontul de start instantaneu, ci cu o incertitudine de până la un eșantion, adică $1/16$ dintr-un bit, ceea ce consumă din start o parte din marjă. Al doilea este că nu un singur eșantion, ci toate trei eșantioanele 8, 9 și 10 trebuie să cadă în interiorul bitului, ceea ce restrânge fereastra utilă. Ținând cont de amândouă, se obțin pentru un cadru cu opt biți de date limitele

$$R_{\text{lent}} = \frac{(D+1)s}{s(D+1) - 1 + s_F} = \frac{9 \cdot 16}{151} = 95,36\%, \quad R_{\text{rapid}} = \frac{(D+2)s}{(D+1)s + s_M} = \frac{10 \cdot 16}{153} = 104,58\%,$$

unde D este numărul de biți de date și de paritate, $s = 16$ este factorul de supraeșantionare, $s_F = 8$ este primul eșantion folosit la vot, iar $s_M = 9$ este eșantionul median. Cu alte cuvinte, ceasul receptorului poate fi cu aproximativ 4,6% mai lent sau mai rapid decât cel al emițătorului înainte ca recepția să se rupă.

Această marjă trebuie însă împărțită între cele două capete, pentru că și emițătorul are eroarea lui, și trebuie să mai rămână loc pentru variația cu temperatura a oscilatoarelor și pentru zgomot. Regula practică folosită în proiectare este ca eroarea fiecărui capăt să nu depășească aproximativ $\pm 2,5\%$, iar eroarea totală acumulată a celor două să rămână sub această valoare. O eroare de câteva procente pare mică, dar acum se vede de ce nu este: ea nu strică un bit, ci deplasează progresiv toate eșantioanele, iar cadrul se pierde integral

atunci când receptorul citește bitul de stop în interiorul ultimului bit de date sau invers. Simptomul tipic, la un terminal serial, este un flux de caractere aparent aleatoare în locul textului așteptat.

8.7 Calculul ratei de transfer

Rata de transfer nu se poate alege liber, pentru că ea se obține prin divizarea întreagă a frecvenței oscilatorului. Blocul care realizează această divizare este un numărător programabil, iar valoarea cu care este încărcat se scrie într-un registru dedicat, numit pe AVR UBRR, de la *USART Baud Rate Register*.

8.7.1 Formula registrului UBRR

Ceasul de eșantionare trebuie să aibă frecvența $s \cdot B$, unde $s = 16$ este factorul de supraeșantionare, iar B este rata de transfer dorită. Divizorul necesar este atunci $f_{osc}/(16B)$. Cum numărătorul divizează prin $UBRR + 1$, valoarea care trebuie scrisă în registru este

$$UBRR = \frac{f_{osc}}{16 \cdot B} - 1,$$

iar rata efectiv obținută, pentru o valoare întreagă a registrului, este

$$B_{ef} = \frac{f_{osc}}{16(UBRR + 1)}.$$

Eroarea relativă rezultată se exprimă în procente ca

$$\varepsilon = \left(\frac{B_{ef}}{B} - 1 \right) \cdot 100\%.$$

Dificultatea apare pentru că rezultatul primei formule nu este, în general, un număr întreg. Registrul acceptă numai valori întregi, deci valoarea calculată trebuie rotunjită, iar rotunjirea introduce o eroare între rata dorită și cea efectiv obținută. Cu cât divizorul este mai mic, adică cu cât rata cerută este mai mare față de frecvența oscilatorului, cu atât o unitate de rotunjire înseamnă o fracțiune mai mare din divizor și deci o eroare relativă mai mare. Acesta este motivul pentru care ratele mari de transfer sunt mult mai greu de generat corect decât cele mici.

Exemplu

Un ATmega328P este alimentat de la un oscilator de 16 MHz. Să se calculeze valoarea registrului UBRR0 și eroarea rezultată pentru ratele de transfer de 9600 și 115200 de bauduri.

Rezolvare. Pentru $B = 9600$:

$$UBRR = \frac{16 \cdot 10^6}{16 \cdot 9600} - 1 = 104,17 - 1 = 103,17 \Rightarrow 103.$$

Rata efectivă este

$$B_{ef} = \frac{16 \cdot 10^6}{16 \cdot 104} = 9615,4 \text{ baudzi},$$

iar eroarea

$$\varepsilon = \left(\frac{9615,4}{9600} - 1 \right) \cdot 100\% = +0,16\%.$$

Valoare cu totul neglijabilă; la 9600 de baudzi legătura este perfect sigură.

Pentru $B = 115200$:

$$\text{UBRR} = \frac{16 \cdot 10^6}{16 \cdot 115200} - 1 = 8,68 - 1 = 7,68 \Rightarrow 8.$$

Rata efectivă devine

$$B_{ef} = \frac{16 \cdot 10^6}{16 \cdot 9} = 111\,111 \text{ baudzi},$$

cu eroarea

$$\varepsilon = \left(\frac{111\,111}{115200} - 1 \right) \cdot 100\% = -3,55\%.$$

Eroarea depășește limita practică de $\pm 2,5\%$, iar dacă și celălalt capăt al legăturii are o eroare de același semn, recepția devine nesigură. Se observă foarte clar diferența: divizorul de 104 tolerează bine rotunjirea, cel de 9 nu.

8.7.2 Modul de viteză dublă

Microcontrolerele AVR oferă o cale parțială de ieșire din această situație prin bitul U2X0 din registrul UCSR0A. Când acest bit este pus pe 1, factorul de supraeșantionare scade de la 16 la 8, iar formula devine

$$\text{UBRR} = \frac{f_{osc}}{8 \cdot B} - 1.$$

Divizorul se dublează, deci rotunjirea are un efect relativ mai mic. Pentru exemplul anterior, la 115200 de baudzi cu oscilator de 16 MHz se obține $\text{UBRR} = 16,36 \Rightarrow 16$, iar rata efectivă devine $16 \cdot 10^6 / (8 \cdot 17) = 117\,647$ de baudzi, cu o eroare de $+2,1\%$. Este o îmbunătățire semnificativă față de $-3,55\%$ și este exact ceea ce face biblioteca Arduino atunci când se cere 115200 de baudzi.

Câștigul nu este însă gratuit. Cu opt eșantioane pe bit în loc de șaisprezece, votul majoritar se face pe eșantioanele 4, 5 și 6, iar fereastra utilă de decizie se îngustează. Marja de eroare tolerată de receptor scade, aproximativ la jumătate, iar imunitatea la zgomot se reduce corespunzător. Modul de viteză dublă este deci un compromis: se plătește robustețe pentru precizia frecvenței.

8.7.3 De ce există cristale de 14,7456 MHz

Un inginer care vede pentru prima dată un cuarț de 14,7456 MHz într-un catalog este tentat să creadă că este o greșeală de tipar. Valoarea este însă aleasă cu grijă, iar motivul

este exact problema discutată mai sus. Se observă că

$$14\,745\,600 = 115\,200 \times 128 = 9\,600 \times 1\,536,$$

deci această frecvență este un multiplu întreg atât al ratei de 115200, cât și al celor de 9600, 19200, 38400 și 57600 de bauzi. Pentru 115200 de bauzi rezultă

$$\text{UBRR} = \frac{14\,745\,600}{16 \cdot 115\,200} - 1 = 8 - 1 = 7,$$

un număr întreg exact, deci eroare zero. Aceeași proprietate o au frecvențele de 11,0592 MHz, 18,432 MHz și 22,1184 MHz, toate multipli ai lui 1,8432 MHz, care nu este altceva decât ceasul de eșantionare pentru 115200 de bauzi.

Aceste cristale, numite uneori *cristale de baud*, sunt alese ori de câte ori legătura serială trebuie să fie sigură la viteză mare. Ele nu se folosesc pe plăcile Arduino, unde s-a preferat o frecvență rotundă de 16 MHz, mai comodă pentru calculele de temporizare ale timerelor, cu prețul erorii de rată de transfer discutate mai sus. Este un exemplu clar de compromis de proiectare: nu există o frecvență care să fie simultan convenabilă pentru temporizări zecimale și pentru rate de transfer standard.

Observație

Valoarea scrisă în registru depinde de frecvența oscilatorului, nu doar de rata de transfer dorită. Un program care funcționează corect pe o placă la 16 MHz și este mutat, fără modificări, pe una la 8 MHz va transmite la jumătate din rata dorită, iar la celălalt capăt nu se va înțelege nimic. La 8 MHz, aceeași rată de 9600 de bauzi cere $\text{UBRR} = 51$, nu 103. Din acest motiv, în codul portabil valoarea nu se scrie niciodată ca o constantă numerică, ci se calculează la compilare, pornind de la macroul `F_CPU`.

8.8 Registrele modului USART al microcontrolerului ATmega328P

Modulul serial al lui ATmega328P este un USART complet, capabil să lucreze atât asincron, cât și sincron; în modul asincron el se comportă exact ca un UART. Fiind un singur modul pe acest microcontroler, registrele lui poartă indicele 0. Structura de registre este comună tuturor UART-urilor și cuprinde trei categorii: un registru de date, registre de control și stare și registrul ratei de transfer.

8.8.1 Registrul de date UDR0

Registrul `UDR0` este locul prin care trec toți octeții, în ambele sensuri. El are o particularitate care surprinde la prima întâlnire: deși are o singură adresă, sunt de fapt două registre fizice distincte. O scriere în `UDR0` încarcă tamponul de emisie, iar o citire din `UDR0` returnează conținutul tamponului de recepție. Nu se poate deci citi înapoi ce s-a scris.

În spatele acestor tampoane se află, în ambele sensuri, câte un registru de deplasare. La emisie, octetul scris în tampon este transferat în registrul de deplasare de îndată ce acesta

este liber, iar de acolo biții ies unul câte unul pe **TxD**. Existența tamponului separat este ceea ce permite programului să pregătească octetul următor în timp ce cel curent este încă pe fir. La recepție, biții recuperați prin votul majoritar intră în registrul de deplasare și, la sfârșitul cadrului, sunt transferați în tamponul de recepție, care pe ATmega328P are două niveluri, formând un mic FIFO.

8.8.2 Registrele de control și stare

Cele trei registre de opt biți **UCSR0A**, **UCSR0B** și **UCSR0C** configurează modulul și raportează starea lui. Tabelul 8.1 rezumă biții cei mai importanți.

Tabela 8.1 Biții importanți ai registrelor de control și stare ale USART0

Registru	Bit	Rol
UCSR0A	RXC0	Recepție completă: un octet este disponibil în tamponul de recepție. Se șterge automat la citirea lui UDRO
	TXC0	Transmisie completă: ultimul bit a părăsit registrul de deplasare și nu mai există date în tampon
	UDRE0	Tamponul de emisie este gol și poate primi un octet nou
	FEO	Eroare de cadru
	DOR0	Depășire a tamponului de recepție
	UPE0	Eroare de paritate
	U2X0	Mod de viteză dublă (supraeșantionare de 8 ori)
UCSR0B	RXCIE0	Validează întreruperea de recepție completă
	TXCIE0	Validează întreruperea de transmisie completă
	UDRIE0	Validează întreruperea de tampon de emisie gol
	RXEN0	Activează receptorul și preia controlul pinului RxD
	TXEN0	Activează emițătorul și preia controlul pinului TxD
	UCSZ02	Al treilea bit al câmpului de dimensiune a cadrului
UCSR0C	UMSEL01:00	Selectează modul: 00 asincron, 01 sincron
	UPM01:00	Paritate: 00 dezactivată, 10 pară, 11 impară
	USBS0	Numărul biților de stop: 0 pentru unul, 1 pentru doi
	UCSZ01:00	Primii doi biți ai câmpului de dimensiune a cadrului
	UCPOL0	Polaritatea ceasului, folosită numai în modul sincron

Dimensiunea cadrului se codifică pe trei biți răspândiți în două registre: **UCSZ02** în **UCSR0B** și **UCSZ01:00** în **UCSR0C**. Combinația 011 corespunde celor opt biți de date folosiți aproape întotdeauna, iar 111 corespunde cadrului de nouă biți, caz în care al nouălea bit se citește și

se scrie separat, prin biții RXB80 și TXB80.

Este util de reținut că activarea receptorului prin RXEN0 și a emițătorului prin TXEN0 preia automat controlul asupra pinilor PD0 și PD1, indiferent de ce s-a configurat anterior în registrul DDRD. Modulul serial are prioritate față de portul de intrare-ieșire de uz general, ceea ce scutește programatorul de configurarea direcției acestor pini, dar explică și de ce pinii digitali 0 și 1 ai plăcii Arduino Uno nu mai pot fi folosiți ca pini obișnuiți odată ce interfața serială a fost pornită.

8.8.3 Indicatorii de eroare

Trei biți din UCSR0A raportează probleme apărute la recepție, iar distincția dintre ei este de mare ajutor la depanare.

Bitul FE0, *frame error*, semnalează o **eroare de cadru**: în poziția în care ar fi trebuit să se afle primul bit de stop, receptorul a citit 0 în loc de 1. Interpretarea cea mai frecventă este că receptorul nu s-a mai aflat, în acel moment, acolo unde credea că se află, adică ratele de transfer ale celor două capete nu coincid. Alte cauze posibile sunt un număr diferit de biți de date configurat la cele două capete sau o întrerupere a legăturii în mijlocul unui cadru.

Bitul DOR0, *data overrun*, semnalează o **depășire**: un cadru nou a fost recepționat complet în timp ce tamponul era deja plin, pentru că programul nu a citit la timp octeții anteriori. Datele se pierd efectiv. Acesta nu este un simptom de eroare electrică, ci de eroare de arhitectură software, iar tratarea lui face obiectul unei secțiuni următoare.

Bitul UPE0, *parity error*, semnalează o **eroare de paritate**: bitul de paritate primit nu corespunde biților de date, deci pe fir s-a schimbat un număr impar de biți. Acesta indică zgomot electric sau o configurare diferită a parității la cele două capete.

Cei trei biți se referă la octetul care se află în vârful tamponului, adică la octetul care va fi returnat de următoarea citire a lui UDR0. De aici rezultă o regulă strictă de programare: registrul UCSR0A trebuie citit *înainte* de UDR0, pentru că citirea registrului de date avansează tamponul și pierde definitiv informația de eroare asociată octetului respectiv.

8.9 Aplicație: transmisia și recepția unui caracter

Pentru a lega toate elementele de mai sus, se cere realizarea unui program care inițializează interfața serială la 9600 de baudzi, în format 8N1, și transmite periodic un caracter, respectiv recepționează caracterele primite. Microcontrolerul este un ATmega328P alimentat de la un oscilator de 16 MHz.

8.9.1 Inițializarea

Inițializarea presupune trei operații: scrierea valorii calculate a divizorului în UBRR0, activarea emițătorului și a receptorului în UCSROB și stabilirea formatului cadrului în UCSROC. Registrul UBRR0 este de 12 biți și este accesibil ca două registre de opt biți, UBRR0H pentru partea superioară și UBRR0L pentru cea inferioară; ordinea de scriere are importanță, partea superioară trebuie scrisă prima.

Exemplu 8.1 Initializarea modului USART0 la 9600 baudzi, format 8N1

```

1 #define F_CPU 16000000UL
2 #include <avr/io.h>
3
4 // Calculul valorii UBRR direct la compilare, cu rotunjire corecta
   ↪ .
5 #define BAUD 9600UL
6 #define UBRR_VAL ((F_CPU + 8UL * BAUD) / (16UL * BAUD) - 1UL)
7
8 void usart_init(void) {
9     // 1. Divizorul ratei de transfer: intai octetul superior
10    UBRR0H = (uint8_t)(UBRR_VAL >> 8);
11    UBRR0L = (uint8_t)(UBRR_VAL & 0xFF);
12
13    // 2. Activarea emitatorului si a receptorului
14    UCSR0B = (1 << RXEN0) | (1 << TXEN0);
15
16    // 3. Formatul cadrului: asincron, 8 biti de date, fara
       ↪ paritate,
17    // un bit de stop (UMSEL = 00, UPM = 00, USBS = 0, UCSZ =
       ↪ 011)
18    UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);
19 }

```

Expresia folosită pentru `UBRR_VAL` merită o explicație. Împărțirea întreagă din limbajul C trunchiază, adică aruncă partea fracționară, ceea ce ar introduce o eroare sistematică într-un singur sens. Adăugarea lui $8 \cdot \text{BAUD}$ la numărător, adică a jumătate din numitor, transformă trunchierea în rotunjire la cel mai apropiat întreg și înjumătățește eroarea maximă. Este un artificiu standard, folosit și în bibliotecile Arduino.

8.9.2 Transmisia și recepția prin interogare

Cea mai simplă formă de utilizare este *interogarea* (polling): programul verifică în buclă indicatorul de stare și acționează atunci când acesta se schimbă.

Exemplu 8.2 Transmisia si receptia unui octet prin interogare

```

1 void usart_transmite(uint8_t data) {
2     // Asteapta golirea tamponului de emisie
3     while (!(UCSR0A & (1 << UDRE0))) {
4         ;
5     }
6     UDR0 = data; // scrierea porneste transmisia
7 }

```

```

8
9 uint8_t usart_receptioneaza(void) {
10     // Asteapta sosirea unui octet complet
11     while (!(UCSR0A & (1 << RXC0))) {
12         ;
13     }
14     return UDR0;    // citirea sterge automat indicatorul RXC0
15 }

```

Cele două funcții sunt simetrice, dar semnificația așteptării este diferită. La emisie, așteptarea este limitată și previzibilă: cel mult durata unui cadru, adică aproximativ o milisecundă la 9600 de baudzi. La recepție, în schimb, așteptarea poate dura la nesfârșit, dacă celălalt capăt nu transmite nimic. O funcție de recepție blocantă oprește deci întregul program pe termen nedefinit, ceea ce este inacceptabil într-o aplicație care mai are și altceva de făcut. Varianta corectă, în astfel de situații, este să se testeze doar disponibilitatea datelor și să se citească numai dacă există:

Exemplu 8.3 Verificarea neblocaanta a disponibilitatii datelor

```

1 uint8_t usart_date_disponibile(void) {
2     return (UCSR0A & (1 << RXC0)) != 0;
3 }
4
5 void bucla_principala(void) {
6     while (1) {
7         if (usart_date_disponibile()) {
8             uint8_t stare = UCSR0A;    // se citește starea
9             // ↪ ÎNAINTE de UDR0
10            uint8_t c = UDR0;
11            if (stare & ((1 << FE0) | (1 << DOR0) | (1 << UPE0)))
12                // ↪ {
13                // cadru eronat: se ignora octetul
14            } else {
15                usart_transmite(c);    // ecou catre emitator
16            }
17        }
18        // aici se pot executa alte sarcini ale aplicatiei
19    }
20 }

```

Se observă respectarea regulii enunțate anterior: registrul de stare este citit înaintea registrului de date, pentru ca indicatorii de eroare să se refere încă la octetul care urmează a fi extras.

Cerința din enunț, transmisia repetată a caracterului 'G' la fiecare jumătate de secundă, se rezolvă apoi într-o singură linie în bucla principală, apelând `usart_transmite('G')` și temporizând între apeluri. Este de preferat ca temporizarea să fie realizată neblokant, cu un timer, așa cum s-a arătat în capitolul dedicat modulelor de temporizare.

8.9.3 Echivalentul cu funcțiile Arduino

Aceleași operații se scriu în mediul Arduino cu clasa `Serial`, care nu este altceva decât un strat subțire peste registrele discutate, la care se adaugă tamponare circulare și rutine de întrerupere.

Exemplu 8.4 Aceeași aplicație folosind clasa `Serial` din Arduino

```

1 void setup() {
2     Serial.begin(9600);           // echivalentul lui usart_init()
3 }
4
5 void loop() {
6     Serial.write('G');             // transmite un singur octet
7     Serial.println(" salut");      // transmite un sir si terminatorul
        ↳ de linie
8
9     if (Serial.available() > 0) {  // cati octeti sunt in
        ↳ tamponul de receptie
10         char c = Serial.read();    // extrage cel mai vechi octet
11         Serial.write(c);           // ecou
12     }
13
14     delay(500);
15 }
```

Diferența de comportare merită subliniată. Funcția `Serial.available()` nu citește indicatorul `RXC0`, ci numărul de octeți aflați într-un tampon circular de 64 de octeți întreținut în memoria RAM de rutina de întrerupere de receptie. Funcția `Serial.write()`, la rândul ei, nu așteaptă golirea tamponului hardware decât dacă tamponul software de emisie este plin; în rest, depune octetul în tampon și revine imediat, transmisia efectivă făcându-se în fundal, prin întreruperi. Aceasta este exact arhitectura descrisă în secțiunea următoare, iar cunoașterea ei explică de ce, la un flux de date rapid și o buclă principală lentă, caracterele se pierd totuși: tamponul de 64 de octeți se umple.

8.10 Recepția cu întrerupere și tamponul circular

Interogarea este simplă și potrivită pentru programe mici, dar are un defect structural: programul trebuie să treacă prin punctul de verificare mai des decât sosesc caracterele. La 115200 de baudzi, un cadru durează aproximativ $87\ \mu\text{s}$, ceea ce înseamnă că bucla principală

are la dispoziție cam 1400 de cicluri de ceas, la 16 MHz, între două caractere. Orice funcție care durează mai mult - o conversie analog-numerică cu așteptare, un apel de afișare pe un ecran LCD, o temporizare cu `delay()` - face ca al doilea caracter sosit între timp să suprascrie tamponul hardware și să ridice indicatorul DOR0. Datele se pierd, iar programul nici măcar nu observă, dacă nu testează explicit indicatorul.

Soluția este să se separe complet momentul *sosirii* datelor de momentul *prelucrării* lor. Sosirea se tratează într-o rutină de întrerupere, care este declanșată de hardware imediat ce un cadru este complet și care nu face decât să mute octetul din registrul UDR0 într-o zonă de memorie. Prelucrarea rămâne în bucla principală, care extrage din acea zonă atunci când are timp. Structura de date care mediază între cele două este un *tampon circular* (buffer circular sau coadă FIFO).

Un tampon circular este un tablou de dimensiune fixă însoțit de doi indici: unul care arată unde se scrie următorul octet sosit, numit *cap*, și unul care arată de unde se citește următorul octet de prelucrat, numit *coadă*. Fiecare indice avansează cu o unitate la fiecare operație și, când ajunge la capătul tabloului, revine la zero. Tamponul este gol când cei doi indici sunt egali și este plin când avansarea capului ar duce la egalarea cozii. Nu există deplasare de date, nu există alocare dinamică, iar ambele operații se execută în timp constant, ceea ce este esențial într-o rutină de întrerupere.

Alegerea unei dimensiuni egale cu o putere a lui doi permite înlocuirea operației de modulo, costisitoare pe un procesor de 8 biți fără divizor hardware, cu o mascare binară.

Exemplu 8.5 Recepție pe întrerupere cu tampon circular

```

1 #include <avr/interrupt.h>
2
3 #define BUF_SIZE 64                // obligatoriu o putere a lui
   ↪ doi
4 #define BUF_MASK (BUF_SIZE - 1)
5
6 static volatile uint8_t buf[BUF_SIZE];
7 static volatile uint8_t cap = 0;    // unde scrie întreruperea
8 static volatile uint8_t coada = 0;  // de unde citește programul
   ↪ principal
9
10 // Rutina de întrerupere: se execută la fiecare cadru recepționat
   ↪ complet
11 ISR(USART_RX_vect) {
12     uint8_t stare = UCSR0A;         // starea ÎNAINTE de citirea
   ↪ datelor
13     uint8_t data = UDR0;            // citirea eliberează tamponul
   ↪ hardware
14

```

```

15     if (stare & ((1 << FE0) | (1 << DOR0) | (1 << UPE0))) {
16         return;                                // cadru eronat, se arunca
17     }
18
19     uint8_t cap_urmator = (cap + 1) & BUF_MASK;
20     if (cap_urmator != coada) {                 // exista loc in tampon
21         buf[cap] = data;
22         cap = cap_urmator;
23     }
24     // altfel tamponul este plin si octetul se pierde controlat
25 }
26
27 uint8_t buffer_gol(void) {
28     return cap == coada;
29 }
30
31 uint8_t buffer_citeste(void) {
32     uint8_t data = buf[coada];
33     coada = (coada + 1) & BUF_MASK;
34     return data;
35 }

```

Activarea acestui mecanism cere doar două linii suplimentare în funcția de inițializare: validarea întreruperii de recepție prin bitul `RXCIE0` și validarea globală a întreruperilor.

Exemplu 8.6 Activarea intreruperii de receptie

```

1  UCSR0B = (1 << RXEN0) | (1 << TXEN0) | (1 << RXCIE0);
2  sei();    // validare globala a intreruperilor

```

Câteva detalii ale acestui cod sunt esențiale și merită argumentate. Cuvântul cheie `volatile` aplicat tabloului și celor doi indici îi spune compilatorului că valorile lor se pot schimba în afara fluxului obișnuit de execuție, deci că nu are voie să le păstreze în registre între utilizări și nici să elimine bucle care par să nu modifice nimic. Fără `volatile`, o buclă de forma `while (buffer_gol()) { }` ar putea fi optimizată într-o buclă infinită, pentru că, din punctul de vedere al compilatorului, nimic din interiorul ei nu schimbă condiția.

Al doilea detaliu este separarea rolurilor: întreruperea scrie numai indicele `cap`, iar programul principal scrie numai indicele `coada`. Fiind variabile de un octet, scrierile lor sunt atomice pe un AVR de 8 biți, deci nu este nevoie de dezactivarea întreruperilor la fiecare acces. Dacă tamponul ar avea mai mult de 256 de poziții, indicii ar deveni de 16 biți, scrierea lor nu ar mai fi atomică, iar accesul din bucla principală ar trebui protejat.

Al treilea detaliu este comportarea la umplere. Codul de mai sus verifică dacă există loc înainte de a scrie și, dacă nu există, aruncă octetul. Alternativa, suprascrierea celui mai vechi

octet, ar strica ordinea mesajelor. Ambele soluții pierd date; diferența este că pierderea este aici controlată și poate fi semnalată aplicației printr-un contor de erori, în loc să se producă în tăcere în hardware.

Observație

O rutină de întrerupere de recepție trebuie să fie cât mai scurtă posibil. Ea nu trebuie să interpreteze mesajul, să afișeze nimic, să apeleze funcții de temporizare sau să transmită la rândul ei. Singura ei sarcină este să golească registrul `UDR0` înainte de sosirea următorului cadru. Orice prelucrare introdusă în rutină reduce marja disponibilă și readuce, pe altă cale, exact problema pe care întreruperea trebuia să o rezolve.

8.11 De la UART la USART

Modulul *USART*, *Universal Synchronous and Asynchronous Receiver Transmitter*, este o extindere a UART-ului care adaugă, pe lângă modul asincron descris până acum, un mod sincron. Diferența constă într-o linie suplimentară de ceas, notată `XCK` pe microcontrolerele AVR, care pe ATmega328P este multiplexată pe pinul `PD4`.

În modul sincron, unul dintre cei doi parteneri generează semnalul de ceas, iar celălalt îl primește; primul se numește master, al doilea slave. Fiecare bit de pe linia de date este validat de un front al acestui ceas, iar bitul `UCPOL0` stabilește pe care dintre fronturi se face eșantionarea, respectiv pe care se modifică datele. Receptorul nu mai are nevoie nici de supraeșantionare, nici de vot majoritar, nici de un ceas propriu precis: el citește linia exact atunci când i se spune.

Avantajele modului sincron decurg direct din această construcție. Riscul de eroare de sincronizare dispare practic, pentru că nu mai există două ceasuri care să se abată unul față de altul; nu mai există nici limitarea impusă de rotunjirea divizorului, pentru că rata este dată de ceasul transmis, oricare ar fi el. Se pot deci atinge viteze mult mai mari, iar cadrul poate renunța la biții de start și de stop, deoarece funcția lor de sincronizare este preluată de ceas, ceea ce crește randamentul transmisiei.

Dezavantajul este la fel de direct: se consumă un fir în plus, iar acel fir trebuie tras pe toată lungimea legăturii. Pe distanțe mari, un fir de ceas suplimentar este scump și, mai ales, sensibil: dacă ceasul culege zgomot, receptorul înregistrează biți inexistenți, ceea ce este mai grav decât o deplasare de fază. Din acest motiv, modul sincron al USART-ului este rar folosit în practică, iar acolo unde se dorește o comunicație sincronă se preferă protocoalele dedicate SPI și I²C, tratate în capitolul următor.

Modulul USART păstrează logica ambelor moduri și le comută prin biții `UMSEL01:00` din `UCSR0C`. Atunci când se lucrează asincron, linia de ceas nu este folosită și rămâne disponibilă ca pin de intrare-ieșire de uz general. Practic, pe ATmega328P se scrie „UART” și se citește „USART în mod asincron”; cele două denumiri desemnează același bloc hardware, iar diferența este doar de configurare.

8.12 Standardele RS-232 și RS-485

Tot ce s-a discutat până acum privește *logica* transmisiei: ordinea biților, formatul cadrului, sincronizarea. UART-ul și USART-ul nu spun însă absolut nimic despre *electrică* legăturii: ce tensiune reprezintă un 1 logic, cât de mare poate fi curentul, cât de lung poate fi cablul, ce conectoare se folosesc. Aceste aspecte fac obiectul unor standarde separate, dintre care cele mai cunoscute sunt RS-232 și RS-485.

8.12.1 Nivelurile TTL/CMOS nu sunt RS-232

Pinii TxD și RxD ai unui ATmega328P lucrează cu nivelurile logice obișnuite ale microcontrollerului: aproximativ 0 V pentru 0 logic și tensiunea de alimentare, 5 V sau 3,3 V, pentru 1 logic. Aceste niveluri se numesc, prin tradiție, niveluri TTL sau CMOS. Ele sunt perfect potrivite pentru legături scurte, de câțiva centimetri, între circuite aflate pe același cablaj sau pe cablaje alăturate.

Standardul RS-232, conceput în anii '60 pentru legături cu modemuri și teleimprimatoare, folosește cu totul altceva. Un 1 logic, numit *mark*, este reprezentat printr-o tensiune *negativă*, între -3 și -15 V, iar un 0 logic, numit *space*, printr-o tensiune pozitivă, între $+3$ și $+15$ V; driverele uzuale lucrează în jur de ± 12 V. Intervalul dintre -3 și $+3$ V este declarat nedefinit și constituie o zonă moartă care conferă imunitate la zgomot.

Se observă două diferențe fundamentale față de nivelurile TTL. Prima este amplitudinea: o excursie totală de aproximativ 24 V în loc de 5 V oferă un raport semnal-zgomot mult mai bun și permite lungimi de cablu mai mari. A doua, mai periculoasă, este **inversarea polarității**: în RS-232, 1 logic este tensiune negativă, exact invers față de convenția TTL. Rezultă că un pin TxD de microcontroller nu poate fi legat direct la un port RS-232 de calculator din două motive independente, fiecare suficient: semnalul ar fi inversat, iar tensiunea negativă de -12 V aplicată unui pin de microcontroller îl distruge aproape sigur.

Rolul unui *convertor de nivel*, cum este binecunoscutul circuit MAX232, este tocmai să facă această traducere în ambele sensuri. El conține, pe de o parte, driver-e care transformă nivelurile de 0 și 5 V în ± 9 sau ± 12 V cu polaritate inversată și, pe de altă parte, receptoare care fac operația inversă, cu limitarea tensiunii la valori sigure pentru microcontroller. Cea mai ingenioasă parte a lui este că generează singur tensiunile negative și pozitive necesare, pornind de la unica alimentare de 5 V, folosind o pompă de sarcină cu patru condensatoare externe.

Observație

Un cablu USB-serial de tipul celor folosite la programarea plăcilor moderne nu conține niveluri RS-232, ci lucrează direct în TTL de 5 V sau 3,3 V, ceea ce îl face compatibil cu pinii microcontrollerului. Confuzia dintre un astfel de cablu și un adaptor USB la RS-232 adevărat, cu conector DE-9, este o cauză frecventă de circuite arse. Înainte de conectare trebuie verificată tensiunea la pinul de emisie al adaptorului, în repaus: aproximativ $+5$ V înseamnă TTL, aproximativ -9 sau -12 V înseamnă RS-232.

8.12.2 RS-485 și semnalul diferențial

RS-232 rezolvă problema amplitudinii, dar păstrează un defect de principiu: nivelul semnalului este definit față de masă, deci este o transmisie *asimetrică* (single-ended). Într-o instalație industrială, masele a două echipamente aflate la sute de metri unul de altul nu sunt la același potențial; diferența poate atinge volți întregi și variază în timp, din cauza curenților care circulă prin structura metalică a clădirii. Această diferență se adaugă direct semnalului util. În plus, orice câmp electromagnetic exterior induce în firul de semnal o tensiune parazită care se suprapune peste informație.

Standardul RS-485 elimină ambele probleme prin *semnalizare diferențială*. Informația nu mai este purtată de tensiunea unui fir față de masă, ci de *diferența* de tensiune dintre două fire, notate de obicei A și B și torsadate între ele. Emițătorul comandă cele două fire în antifază: pentru un nivel logic, A este mai pozitiv decât B; pentru celălalt, invers. Receptorul nu măsoară decât diferența $U_A - U_B$ și decide în funcție de semnul ei, pragul tipic fiind de ± 200 mV.

Câștigul este considerabil, iar mecanismul merită înțeles. Cele două fire fiind torsadate, ele parcurg practic același traseu și sunt expuse aceluiași câmp perturbator. Orice tensiune indusă din exterior apare deci aproape identic pe amândouă, ca o tensiune de *mod comun*, și se anulează atunci când receptorul face diferența. La fel se anulează și diferența de potențial dintre masele celor două echipamente, atât timp cât ea rămâne în domeniul de mod comun admis, care în RS-485 este de la -7 la $+12$ V. Aceasta este rațiunea profundă pentru care semnalul diferențial merge mult mai departe decât cel asimetric.

Rezultatul practic este că RS-485 permite legături de până la aproximativ 1200 m, cu până la 32 de dispozitive pe aceeași pereche de fire, la viteze de ordinul zecilor sau sutelor de kilobiți pe secundă la distanță maximă, respectiv de ordinul megabiților pe distanțe scurte. Există un compromis clar între lungime și viteză, pentru că atenuarea și dispersia cablului cresc cu frecvența. Comunicația este, în varianta uzuală cu două fire, semiduplex: aceeași pereche servește ambele sensuri, iar fiecare emițător trebuie explicit activat și apoi dezactivat, printr-un pin de comandă al circuitului transceiver, astfel încât linia să rămână liberă pentru ceilalți participanți.

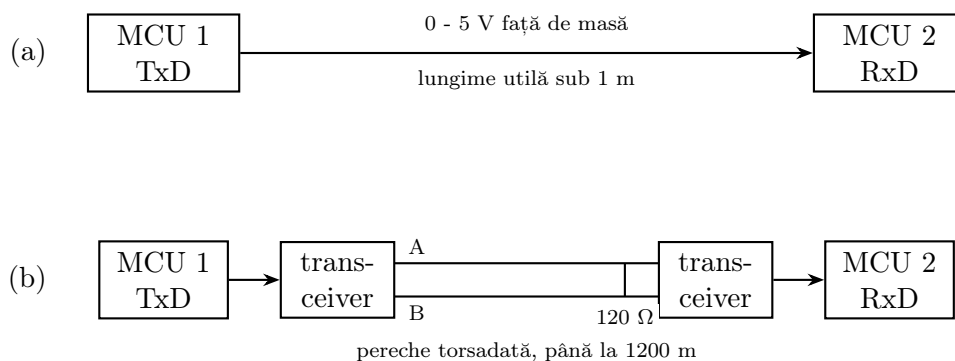


Figura 8.3 Comparatie între o legătură serială directă cu niveluri TTL (a) și una prin RS-485, cu semnal diferențial pe pereche torsadată și terminație de adaptare (b).

8.12.3 Terminarea liniei

La lungimile la care lucrează RS-485, cablul nu mai poate fi tratat ca un simplu conductor, ci trebuie privit ca o *linie de transmisie* cu o impedanță caracteristică proprie, în general de aproximativ $120\ \Omega$ pentru perechea torsadată folosită. Un front de tensiune care se propagă pe această linie și ajunge la un capăt neadaptat, adică terminat pe o impedanță mult diferită de cea caracteristică, nu dispare, ci se reflectă și se întoarce spre emițător. Unda reflectată se suprapune peste semnalul util și produce supracreșteri și oscilații care pot fi interpretate de receptor ca tranziții suplimentare.

Remediul este *terminarea liniei*: la fiecare dintre cele două capete fizice ale magistralei se conectează, între A și B, un rezistor egal cu impedanța caracteristică, adică de $120\ \Omega$. Energia unei incidente este atunci absorbită în rezistor în loc să fie reflectată. Se termină numai cele două capete, nu și dispozitivele intermediare; un rezistor de terminare montat la mijlocul magistralei nu ajută cu nimic și doar încarcă inutil emițătorul. Efectul reflexiilor devine sesizabil atunci când durata de creștere a frontului devine comparabilă cu timpul de propagare dus-întors pe cablu, ceea ce explică de ce, la viteze mici și distanțe scurte, o magistrală neterminată poate funcționa aparent corect, iar la creșterea vitezei se defectează brusc.

Tabelul 8.2 rezumă comparativ caracteristicile celor două standarde.

Tabela 8.2 Comparatie între standardele RS-232 și RS-485

Caracteristică	RS-232	RS-485
Topologie	Punct la punct	Multipunct (magistrală)
Tip de semnal	Asimetric, față de masă	Diferențial, pe pereche torsadată
Distanță maximă	Aproximativ 15 m	Aproximativ 1200 m
Număr de dispozitive	2	Până la 32 de sarcini unitare
Imunitate la zgomot	Mică	Mare
Tensiuni de lucru	$\pm 12\text{ V}$ tipic	$\pm 5\text{ V}$ diferențial
Mod de lucru	Duplex integral	Semiduplex, uzual cu două fire

Merită subliniat, la final, că ambele standarde descriu numai stratul fizic. Formatul cadrului rămâne cel al UART-ului: bit de start, biți de date, eventual paritate, bit de stop. Un microcontroler care transmite prin RS-485 folosește exact același modul USART și exact aceleași registre ca într-o legătură TTL directă; se schimbă doar circuitul de interfață conectat la pinii TxD și RxD, plus un pin de uz general care comandă activarea emițătorului. Această separare între protocol și nivel electric este unul dintre motivele longevității remarcabile a comunicației seriale asincrone.

8.13 Concluzii

Comunicația serială asincronă rezolvă, cu un minimum de resurse hardware, problema transferului de informație între două circuite independente. Renunțarea la firul de ceas se plătește printr-o convenție prealabilă asupra vitezei și printr-un mecanism de resincronizare la fiecare cadru, bazat pe bitul de start. Receptorul supraeșantionează linia de 16 ori pe bit, decide prin vot majoritar în jurul mijlocului fiecărui bit și tolerează o abatere de ceas de câteva procente, cu condiția ca aceasta să nu se acumuleze peste jumătate de bit până la sfârșitul cadrului.

Din această constrângere decurge întreaga problematică a calculului ratei de transfer. Divizorul întreg impus de registrul UBRR face ca unele combinații de frecvență a oscilatorului și rată de transfer să fie exacte, iar altele să introducă erori inacceptabile; de aici existența cristalelor de 14,7456 MHz și a modului de viteză dublă.

La nivel de programare, modulul USART al lui ATmega328P se reduce la un registru de date, trei registre de control și stare și un registru de divizor. Recepția prin interogare este suficientă pentru programe simple, dar arhitectura corectă pentru fluxuri rapide combină o rutină de întrerupere scurtă cu un tampon circular, separând sosirea datelor de prelucrarea lor.

În sfârșit, protocolul și nivelul electric sunt lucruri distincte. Aceleași cadre pot circula la niveluri TTL pe câțiva centimetri, prin RS-232 pe cincisprezece metri sau prin RS-485, diferențial, pe mai bine de un kilometru, în funcție de circuitul de interfață ales. Capitolul următor va trata cealaltă familie de protocoale seriale, cele sincrone, unde firul de ceas este readus în schemă pentru a obține viteze mult mai mari pe distanțe scurte.

Întrebări de verificare

1. Explicați de ce comunicația serială a înlocuit-o pe cea paralelă în aproape toate aplicațiile moderne, deși transmite un singur bit la un moment dat. Care este rolul decalajului de propagare între firele unei magistrale paralele și de ce efectul lui se agravează odată cu creșterea frecvenței?
2. Ce anume trebuie să cunoască dinainte receptorul într-o legătură serială asincronă și de ce această informație nu poate fi dedusă din semnalul primit? Argumentați pornind de la codificarea NRZ.
3. Care este rolul bitului de start și care este rolul bitului de stop? Explicați de ce linia este menținută în 1 logic în starea de repaus și ce s-ar întâmpla dacă starea de repaus ar fi 0 logic.
4. Descrieți mecanismul de supraeșantionare de 16 ori folosit de receptorul AVR. De ce se folosesc eșantioanele 8, 9 și 10 și de ce decizia se ia prin vot majoritar, în loc să se citească un singur eșantion la mijlocul bitului?
5. Un cadru este de tip 8E2, adică opt biți de date, paritate pară și doi biți de stop.

- Calculați la câte durate de bit după frontul de start se află mijlocul ultimului bit de stop și deduceți eroarea relativă maximă de ceas tolerată din considerente pur geometrice. Comparați cu limita obținută pentru 8N1 și explicați diferența.
6. Un ATmega328P alimentat de la un oscilator de 16 MHz trebuie să comunice la 57600 de baudzi. Calculați valoarea registrului UBRRO, rata efectiv obținută și eroarea procentuală, atât în mod normal, cât și cu bitul U2X0 activat. Care dintre cele două variante este preferabilă și de ce?
 7. Explicați de ce eroarea relativă de rată de transfer crește atunci când crește rata cerută, la aceeași frecvență a oscilatorului. Ce proprietate trebuie să aibă frecvența cuarțului pentru ca eroarea să fie nulă la 115200 de baudzi?
 8. Un student citește registrul UDR0 și abia apoi verifică biții de eroare din UCSROA, constatând că nu detectează niciodată erori de cadru. Explicați de ce ordinea celor două operații este greșită și ce se întâmplă intern la citirea lui UDR0.
 9. Distingeți între indicatorii FEO, DOR0 și UPEO. Pentru fiecare, indicați cauza cea mai probabilă și acțiunea corectivă pe care ar trebui să o întreprindă proiectantul.
 10. Un dispozitiv transmite continuu la 115200 de baudzi, iar bucla principală a programului conține un apel `delay(2)`. Estimați câte cadre sosesc în acest interval și explicați ce se întâmplă cu ele în absența unui tampon circular. Cum modifică situația o rutină de întrerupere de recepție?
 11. Argumentați de ce dimensiunea unui tampon circular se alege ca putere a lui doi și de ce variabilele care indexează tamponul trebuie declarate *volatile*. Ce s-ar putea întâmpla în absența acestui calificativ?
 12. Doi pini TxD de microcontroler, la 5 V, sunt legați direct printr-un cablu de zece metri și comunicația este nesigură. Explicați care sunt cauzele fizice și comparați două soluții posibile: trecerea la RS-232 și trecerea la RS-485. Ce anume anulează, în cazul semnalului diferențial, atât perturbațiile induse, cât și diferența de potențial dintre masele celor două echipamente?
 13. Explicați rolul rezistoarelor de terminatie de $120\ \Omega$ pe o magistrală RS-485. De ce se montează exact două, la capetele fizice ale liniei, și de ce o magistrală neterminată poate funcționa corect la viteză mică și se defectează la creșterea vitezei?

9. Magistralele sincrone: SPI și I²C

- De ce este nevoie de o magistrală sincronă
- Magistrala I²C: prezentare generală
- Principiul de funcționare al magistralei I²C
- Tranzacții pe magistrala I²C
- Magistrala SPI: principiul de funcționare
- Transferul datelor și modurile SPI
- Topologii SPI
- Modulul SPI al microcontrolerului ATmega328P
- Aplicații cu bibliotecile SPI și Wire
- Comparație între UART, SPI și I²C
- Concluzii
- Întrebări de verificare

9.1 De ce este nevoie de o magistrală sincronă

Capitolul precedent s-a încheiat cu o interfață care funcționează bine și care, din acest motiv, este încă folosită după mai bine de o jumătate de secol: comunicația serială asincronă de tip UART. Două fire, un bit de start, câțiva biți de date, un bit de stop, și două dispozitive care nu împart nimic altceva decât masa pot schimba informație. Simplitatea aceasta are însă un preț, iar prețul devine vizibil de îndată ce aplicația crește. Un sistem real nu are un singur partener de dialog: are un senzor de temperatură, un accelerometru, un ceas de timp real, o memorie EEPROM externă, un afișaj și, poate, un card de memorie. Întrebarea firească este dacă interfața asincronă poate duce toată această încărcătură. Răspunsul este că nu, și merită înțeles exact de ce.

Prima limitare este *absența unui ceas comun*. Într-o legătură asincronă, emițătorul și receptorul nu împart niciun semnal de sincronizare; fiecare are propriul oscilator și propriul divizor de frecvență, iar singurul lucru asupra căruia s-au înțeles dinainte este rata de transfer nominală. Receptorul se agață de frontul descrescător al bitului de start, pornește un numărător intern și eșantionează linia în momentele în care presupune că se află mijlocul fiecărui bit. Presupunerea este corectă doar atât timp cât cele două ceasuri nu se îndepărtează prea mult unul de celălalt. Eroarea se acumulează bit după bit, iar dacă abaterea relativă dintre cele două rate depășește aproximativ două procente și jumătate, ultimul bit

al cadrului este eșantionat deja în afara ferestrei sale, iar octetul recepționat este greșit. Acesta este motivul pentru care capitolul anterior a insistat asupra calculului registrului de rată: nu este o formalitate, ci condiția de existență a legăturii.

A doua limitare decurge din prima: *viteza este mărginită de precizia oscilatoarelor*, nu de electronica liniei. Cu cât rata crește, cu atât fereastra fiecărui bit se îngustează, iar aceeași abatere relativă a ceasului consumă o fracțiune mai mare din ea. Practic, o legătură UART pe un microcontroler cu rezonator ceramic funcționează comod la 9600 sau 115200 de biți pe secundă, dar devine nesigură cu mult înainte de a se apropia de un megabit pe secundă. Or, un afișaj grafic care trebuie reîmprospătat de treizeci de ori pe secundă, sau o memorie de tip card SD din care se citește un fișier, cer ordine de mărime peste această limită. Dacă receptorul ar dispune de ceasul emițătorului, problema ar dispărea cu totul: nu ar mai trebui să ghicească unde este mijlocul bitului, pentru că i s-ar spune, prin chiar semnalul de tact, când anume linia de date este valabilă.

A treia limitare este *numărul de dispozitive*. Legătura asincronă este, prin construcție, punct la punct: o linie de emisie a unui dispozitiv merge la linia de recepție a celuilalt și invers. Nu există niciun mecanism de adresare, deci nu există nicio modalitate prin care trei dispozitive legate la aceleași două fire să știe căruia dintre ele îi este destinat un octet. Microcontrolerul ATmega328P are un singur modul USART. Dacă fiecare periferic ar cere o legătură asincronă proprie, ar trebui fie multiplicat modulele hardware, fie implementate în software mai multe interfețe seriale, cu costul unei încărcări enorme a procesorului. Rezultatul practic este cel din Figura conceptuală a oricărui proiect care crește necontrolat: nu mai sunt suficienți pini de intrare-ieșire de uz general pentru a lega tot ce trebuie legat.

Aceasta este de fapt problema centrală. Un ATmega328P are douăzeci de pini utilizabili ca intrări-ieșiri digitale, iar din ei o parte sunt deja ocupați de oscilator, de programator, de LED-uri sau de butoane. Dacă fiecare periferic nou consumă doi, trei sau opt pini, capacitatea de extindere se epuizează după câteva module. Soluția nu este un microcontroler cu mai mulți pini, ci un principiu diferit de organizare: în loc de o legătură dedicată pentru fiecare periferic, o *magistrală* - un set redus de fire pe care toate dispozitivele le împart, împreună cu un protocol care spune cine vorbește, când vorbește și cui i se adresează.

Cele două magistrale seriale sincrone folosite aproape universal în sistemele încorporate rezolvă exact această problemă, dar cu compromisuri diferite. Magistrala I²C reduce numărul de fire la minimul absolut, două, și plătește pentru asta cu viteză mai mică și cu un protocol mai complicat. Magistrala SPI merge pe drumul opus: acceptă patru fire și un protocol aproape inexistent, în schimbul unei viteze cu un ordin de mărime mai mari. Ambele sunt *sincrone*, adică transportă explicit un semnal de ceas alături de date, ceea ce elimină dintr-o lovitură problema abaterii oscilatoarelor și, odată cu ea, limita superioară de viteză impusă de aceasta.

9.2 Magistrala I²C: prezentare generală

Denumirea I²C vine de la *Inter-Integrated Circuit*, adică o magistrală gândită pentru legarea circuitelor integrate aflate pe aceeași placă sau la distanțe de ordinul zecilor de

centimetri. A fost dezvoltată de firma Philips la începutul anilor optzeci, pentru a reduce numărul de trasee dintre microcontrolerul unui televizor și circuitele pe care acesta trebuia să le comande: selectorul de canale, procesorul de sunet, memoria de setări. Problema era exact cea descrisă mai sus, iar soluția aleasă a fost radicală: două fire, indiferent câte circuite se leagă la ele.

Cele două linii sunt *SDA* (*Serial Data Line*), pe care circulă datele, și *SCL* (*Serial Clock Line*), pe care circulă semnalul de ceas. La ele se adaugă, evident, masa comună, fără de care nivelurile de tensiune nu ar avea sens; masa nu se numără însă printre firele magistralei, pentru că este oricum necesară. Faptul că sunt doar două linii de semnal, și că acest număr rămâne același fie că sunt legate două dispozitive, fie că sunt legate douăsprezece, este marele avantaj al acestei magistrale.

Protocolul este de tip maestru-sclav. *Maestrul* (în engleză *master*) este dispozitivul care generează semnalul de ceas, care inițiază fiecare transfer și care decide când transferul se încheie; în majoritatea aplicațiilor, maestrul este microcontrolerul. *Sclavii* (*slave*) sunt perifericele: senzori, memorii, ceasuri de timp real, extensoare de porturi. Un sclav nu vorbește niciodată nesolicitat; el răspunde numai atunci când maestrul îi cere. Standardul permite și configurații cu mai mulți maestri pe aceeași magistrală, situație în care este nevoie de un mecanism de arbitraj, la care vom reveni; în practica obișnuită a proiectelor cu Arduino, magistrala are un singur maestru și mai mulți sclavi.

Deosebirea esențială față de UART, și motivul pentru care două fire sunt de ajuns pentru oricâte dispozitive, este că fiecare sclav are o *adresă* proprie. Fiecare transfer începe cu emiteria acestei adrese pe magistrală. Toți sclavii ascultă, toți compară adresa primită cu adresa lor proprie, și numai acela care recunoaște propria adresă continuă dialogul; ceilalți rămân tăcuți până la sfârșitul transferului. Adresa este, așadar, mecanismul care înlocuiește firul dedicat: în loc să selectăm dispozitivul prin hardware, îl selectăm prin conținutul mesajului.

Vitezele standardizate ale magistralei formează câteva trepte. Modul standard lucrează la 100 kHz, modul rapid la 400 kHz, iar variantele mai recente ale specificației definesc 1 MHz și 3,4 MHz. Marea majoritate a perifericelor ieftine folosite în laborator, precum senzorii de temperatură și presiune sau ceasurile de timp real, funcționează la 100 kHz sau 400 kHz. Trebuie remarcat că, spre deosebire de UART, aceste frecvențe sunt maxime, nu valori impuse: maestrul poate genera un ceas mai lent fără ca nimic să se strice, pentru că sclavii nu presupun nimic despre ritmul lui. Aceasta este o consecință directă a faptului că ceasul este transmis, nu presupus.

9.3 Principiul de funcționare al magistralei I²C

Dacă mai multe dispozitive sunt legate la același fir, apare imediat o întrebare de natură electrică: ce se întâmplă dacă unul dintre ele vrea să pună linia în starea de unu logic, adică o conectează la tensiunea de alimentare, iar altul vrea să o pună în zero logic, adică o conectează la masă? Răspunsul este că se formează un scurtcircuit prin cele două etaje de ieșire, curentul crește necontrolat, iar unul dintre circuite se distruge. Într-o magistrală la care se leagă un număr oarecare de dispozitive, o astfel de coliziune electrică nu poate fi

exclusă prin convenție software; trebuie făcută imposibilă prin construcție.

9.3.1 Colectorul deschis și rezistențele de ridicare

Soluția adoptată de I²C este ca niciun dispozitiv să nu poată forța linia în starea de unu logic. Etajul de ieșire al fiecărui dispozitiv este de tip *colector deschis* (sau *dren deschis*, la tehnologia MOS): el conține un singur tranzistor, legat între linie și masă. Când tranzistorul conduce, linia este trasă la masă, deci în zero logic. Când tranzistorul este blocat, ieșirea nu mai este conectată nicăieri, iar dispozitivul se comportă ca și cum ar fi absent de pe magistrală; spunem că *eliberează* linia.

Dacă toate dispozitivele își eliberează ieșirile, linia rămâne în aer, iar nivelul ei ar fi nedefinit. De aceea, fiecare dintre cele două linii este legată la tensiunea de alimentare printr-un rezistor, numit *rezistor de ridicare* (*pull-up*). Rolul lui este să readucă linia în starea de unu logic atunci când nimeni nu o trage în jos. Rezultatul acestei construcții este o funcție logică ȘI cablată: linia este în unu logic numai dacă toate dispozitivele o eliberează, și trece în zero logic dacă măcar unul dintre ele o trage la masă. Coliziunea electrică devine imposibilă, pentru că nu există două surse care să se opună; există cel mult mai multe tranzistoare care trag în aceeași direcție. Figura 9.1 arată structura magistralei.

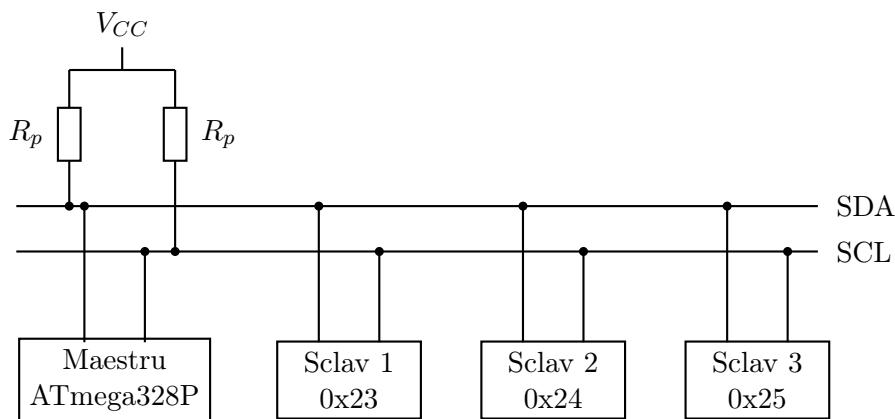


Figura 9.1 Structura unei magistrale I²C: două linii comune, SDA și SCL, ridicate la tensiunea de alimentare prin câte un rezistor, la care se leagă maestrul și oricâți sclavi, fiecare cu adresa lui proprie.

Valoarea rezistoarelor de ridicare nu este indiferentă și rezultă dintr-un compromis. Linia, împreună cu toate capacitățile parazite ale pinilor și ale traseelor legate la ea, formează un circuit de tip rezistor-condensator. Tranziția din zero în unu nu este instantanee, ci exponențială, cu o constantă de timp egală cu produsul dintre rezistența de ridicare și capacitatea totală a magistralei. Un rezistor mare limitează consumul de curent atunci când linia este trasă la masă, dar încetinește frontul crescător; dacă acesta nu ajunge la nivelul de unu logic înainte de următorul front de ceas, datele se citesc greșit. Un rezistor mic accelerează tranziția, dar crește curentul absorbit prin tranzistorul care trage linia la masă, curent care nu poate depăși valoarea admisă de acesta. În practică se folosesc valori de ordinul a 4,7 kΩ pentru magistrale scurte la 100 kHz și de ordinul a 2,2 kΩ pentru 400 kHz sau pentru

magistrale cu mai multe dispozitive. Specificația limitează capacitatea totală a magistralei la 400 pF, ceea ce înseamnă, practic, trasee scurte și un număr rezonabil de dispozitive.

Observație

Pe placa Arduino Uno, liniile SDA și SCL sunt aduse pe pinii A4 și A5, adică pe PC4 și PC5. Multe module comerciale, cum sunt cele cu senzori sau cu afișaje, au deja rezistoarele de ridicare montate pe placa lor. Dacă se leagă mai multe astfel de module pe aceeași magistrală, rezistoarele ajung în paralel, iar rezistența echivalentă scade; cu patru sau cinci module cu câte 4,7 k Ω fiecare, rezistența echivalentă coboară sub un kilohm, curentul crește și magistrala poate deveni instabilă. În asemenea situații se îndepărtează rezistoarele de pe toate modulele în afară de unul.

9.3.2 Adresarea pe șapte biți

Fiecare sclav de pe magistrală este identificat printr-o adresă pe șapte biți, ceea ce oferă în principiu 128 de combinații. Nu toate sunt utilizabile: opt adrese de la începutul intervalului și opt de la sfârșitul lui sunt rezervate de specificație pentru funcții speciale, cum ar fi apelul general adresat tuturor dispozitivelor sau extensia la adresare pe zece biți. Rămân 112 adrese disponibile, ceea ce este mai mult decât suficient pentru orice aplicație obișnuită.

Adresa nu este aleasă de proiectant, ci este stabilită de producătorul circuitului integrat și scrisă în foaia de catalog. Un senzor de temperatură și umiditate poate avea adresa fixă 0x44, un ceas de timp real adresa 0x68, un extensor de porturi o adresă din intervalul 0x20 la 0x27. Multe circuite au unul, doi sau trei pini de configurare a adresei, legați la masă sau la alimentare de către utilizator, care selectează biții cei mai puțin semnificativi ai adresei; în felul acesta se pot conecta pe aceeași magistrală mai multe exemplare ale aceluiași circuit. Limitarea rămâne însă reală: dacă un circuit are adresă fixă, nu putem avea două bucăți din el pe aceeași magistrală, oricâți pini liberi ar avea microcontrolerul.

Cei șapte biți de adresă sunt urmați, în cadrul care se emite pe magistrală, de un al optulea bit care indică sensul transferului: zero pentru scriere, adică maestrul urmează să trimită date sclavului, unu pentru citire, adică maestrul urmează să primească date de la sclav. Împreună, adresa și bitul de sens formează un octet complet, ceea ce explică o confuzie foarte răspândită. Aceeași adresă apare uneori în documentație ca 0x68 și alteori ca 0xD0, pentru că prima este adresa pe șapte biți, iar a doua este octetul emis pe magistrală, adică adresa deplasată cu o poziție la stânga. Bibliotecile Arduino folosesc convenția pe șapte biți și fac ele deplasarea; alte biblioteci cer octetul complet. Consecința unei confuzii de acest fel este că dispozitivul pur și simplu nu răspunde.

9.3.3 Condițiile de start și de stop

Rămâne de rezolvat problema delimitării mesajelor. Într-un cadru UART, începutul este marcat de bitul de start, adică de o tranziție a liniei de date. Pe I²C, o tranziție a liniei de date nu poate marca începutul, pentru că liniile de date se schimbă la fiecare bit. Este nevoie de un semnal care să nu poată fi confundat cu nimic din traficul obișnuit, iar soluția

folosește ambele linii deodată.

Regula generală a protocolului este că *linia SDA are voie să își schimbe starea numai atunci când SCL este în zero logic*. Cât timp SCL este în unu logic, datele trebuie să rămână stabile, pentru că exact atunci le eșantionează receptorul. Prin urmare, o tranziție a lui SDA în timp ce SCL este sus este, prin definiție, ceva ce nu poate apărea în timpul transportului de date; poate fi folosită, deci, ca semnal de control.

Condiția de start este o tranziție a lui SDA din unu în zero în timp ce SCL este menținut în unu. Ea semnalează tuturor dispozitivelor că magistrala tocmai a fost ocupată și că octetul care urmează este o adresă. *Condiția de stop* este tranziția inversă, a lui SDA din zero în unu în timp ce SCL este menținut în unu. Ea eliberează magistrala și o readuce în starea de repaus, în care ambele linii sunt în unu logic. Există și o *condiție de start repetat*, identică cu cea de start, dar emisă fără o condiție de stop înainte; ea permite maestrului să schimbe sensul transferului fără a elibera magistrala între timp, ceea ce, într-un sistem cu mai mulți maștri, împiedică un alt maestru să se strecoare între cele două jumătăți ale operației.

9.3.4 Confirmarea ACK și NACK

Ultima piesă a mecanismului este confirmarea. Spre deosebire de UART, unde emițătorul aruncă octetul pe linie fără să aștepte vreodată dacă a ajuns undeva, I²C cere o confirmare după fiecare octet transmis. Fiecare octet ocupă nouă perioade de ceas, nu opt: opt pentru biții de date, transmiși cu cel mai semnificativ bit primul, și una a noua pentru bitul de confirmare.

Mecanismul se sprijină din nou pe colectorul deschis. În a noua perioadă de ceas, dispozitivul care a transmis octetul eliberează linia SDA, adică nu mai trage de ea. Dacă receptorul a primit octetul și îl acceptă, el trage linia la masă pe durata acestei perioade; emițătorul, care citește linia, vede zero logic și interpretează aceasta ca *ACK*, adică o confirmare. Dacă receptorul nu face nimic, rezistorul de ridicare menține linia în unu logic, iar emițătorul citește *NACK*, adică o neconfirmare.

Semnificația unui NACK depinde de context, iar această dublă semnificație este esențială pentru înțelegerea protocolului. După octetul de adresă, un NACK înseamnă că niciun sclav de pe magistrală nu poartă acea adresă: dispozitivul lipsește, este nealimentat, este legat greșit sau are altă adresă decât cea presupusă. Acesta este, de altfel, principiul pe care se bazează programul de scanare a magistralei pe care îl vom scrie mai târziu. După un octet de date trimis de maestru unui sclav, un NACK înseamnă că sclavul nu poate prelua mai multe date, de exemplu pentru că bufferul lui intern este plin. Iar la citire, lucrurile stau invers: maestrul este cel care confirmă fiecare octet primit, iar el trimite deliberat NACK după ultimul octet pe care îl dorește, ca semnal pentru sclav că trebuie să se oprească din transmis. Un maestru care ar confirma și ultimul octet ar spune sclavului să continue.

Figura 9.2 arată desfășurarea în timp a acestor elemente într-o tranzație de scriere.

Se observă în figură regula fundamentală: SDA se modifică întotdeauna în intervalele în care SCL este jos, cu singurele două excepții de la începutul și sfârșitul tranzației, care sunt tocmai condițiile de start și de stop.

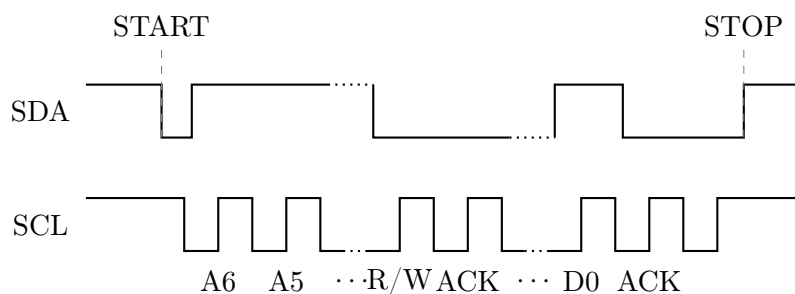


Figura 9.2 Cronograma unei tranzații de scriere pe magistrala I²C: condiția de start, adresa pe șapte biți urmată de bitul de sens, confirmarea sclavului, octetul de date cu confirmarea lui și condiția de stop. Zonele punctate corespund biților omiși din desen.

9.4 Tranzacții pe magistrala I²C

Elementele descrise până acum se combină în două tipare de tranzație, pe care merită să le urmărim pas cu pas, pentru că orice bibliotecă I²C nu face altceva decât să le execute.

9.4.1 Tranzacția de scriere

Să presupunem că maestrul vrea să scrie valoarea 0x1F în registrul cu numărul 0x07 al unui sclav aflat la adresa 0x68. Se întâmplă următoarele. Maestrul verifică mai întâi că magistrala este liberă, adică ambele linii sunt în unu logic, apoi generează condiția de start trăgând SDA la masă în timp ce SCL este încă sus. Emite apoi octetul format din cele șapte biți de adresă și din bitul de sens pus pe zero, adică valoarea 0xD0, câte un bit pe fiecare perioadă de ceas, cu bitul cel mai semnificativ primul. În a noua perioadă eliberează SDA și citește linia: sclavul cu adresa 0x68 a recunoscut adresa și trage linia la masă, deci maestrul citește ACK și știe că are cu cine vorbi.

Urmează al doilea octet, 0x07, care nu este pentru protocol decât o dată oarecare, dar pe care sclavul îl interpretează, conform propriei sale documentații, ca fiind numărul registrului asupra căruia se va lucra. Sclavul confirmă din nou. Al treilea octet, 0x1F, este valoarea propriu-zisă, pe care sclavul o înscrie în registrul selectat anterior și pe care o confirmă la rândul ei. Maestrul nu mai are nimic de trimis, așa că generează condiția de stop, eliberând SDA în timp ce SCL este sus, iar magistrala revine în repaus.

Merită subliniat că protocolul I²C nu știe nimic despre „registre”. Noțiunea de registru intern al sclavului, cu numărul lui, aparține dispozitivului, nu magistralei. Pentru magistrală, transferul de mai sus este pur și simplu: start, adresă cu scriere, trei confirmări, doi octeți de date, stop. Faptul că primul octet de date este interpretat ca adresă de registru și al doilea ca valoare este o convenție respectată de aproape toate perifericele, dar ea ține de foaia lor de catalog.

9.4.2 Tranzacția de citire

Citirea este mai subtilă, tocmai pentru că, de obicei, trebuie mai întâi să spunem sclavului *ce vrem să citim*, iar asta înseamnă o scriere. Secvența obișnuită are două faze. În prima fază,

maestrul generează start, emite adresa cu bitul de sens pe zero, primește confirmarea, trimite numărul registrului 0x07 și primește confirmarea. În acest moment sclavul și-a poziționat pointerul intern pe registrul cerut, dar nu a trimis încă nimic.

În a doua fază, maestrul generează o condiție de *start repetat*, fără să fi emis o condiție de stop, și reemite adresa 0x68, de data aceasta cu bitul de sens pus pe unu, adică octetul 0xD1. Sclavul confirmă și, de acum înainte, rolurile pe linia SDA se inversează: sclavul devine emițător, iar maestrul receptor. Maestrul continuă totuși să genereze ceasul, pentru că el rămâne maestru; ritmul transferului îi aparține în continuare. La fiecare perioadă de ceas, sclavul pune pe SDA câte un bit, iar maestrul îl citește.

După fiecare octet primit, maestrul este cel care confirmă. Dacă mai vrea date, trage SDA la masă în a noua perioadă, adică trimite ACK, iar sclavul continuă cu octetul următor, luat de obicei din registrul cu numărul imediat superior. Dacă octetul primit a fost ultimul dorit, maestrul lasă linia sus, adică trimite NACK, iar sclavul înțelege că trebuie să înceteze și își eliberează definitiv ieșirea. Abia atunci maestrul generează condiția de stop. Ordinea aceasta este obligatorie: dacă maestrul ar emite stop imediat după un ACK, sclavul ar putea rămâne cu ieșirea activă, ținând SDA la masă, iar magistrala ar fi blocată.

Observație

Blocarea magistralei descrisă mai sus este una dintre cele mai neplăcute defecțiuni ale acestei interfețe. Dacă microcontrolerul se resetează în mijlocul unei citiri, sclavul rămâne convins că este în plină transmisie și continuă să țină SDA la masă în așteptarea ceasului. Linia SDA fiind blocată, maestrul nu mai poate genera nici măcar o condiție de start, iar magistrala este moartă până la oprirea alimentării. Procedura de deblocare constă în generarea manuală, prin software, a până la nouă impulsuri pe linia SCL, ceea ce permite sclavului să termine octetul început și să elibereze SDA, urmate de o condiție de stop.

9.4.3 Arbitrajul și întinderea ceasului

Două mecanisme suplimentare completează protocolul și amândouă decurg tot din proprietatea de ȘI cablat a magistralei.

Primul este *arbitrajul*, necesar atunci când pe magistrală există mai mulți maestri și doi dintre ei încep un transfer în același timp. Fiecare emite biții adresei și, în același timp, citește linia SDA. Atâta vreme cât nivelul citit coincide cu cel emis, maestrul continuă. În clipa în care unul dintre ei emite un unu logic, adică eliberează linia, dar citește un zero, înseamnă că altcineva trage de linie; el a pierdut arbitrajul, se retrage imediat din transfer și trece în regim de ascultare. Mecanismul este elegant pentru că nu pierde niciun bit: maestrul câștigător nu observă nimic și își continuă transferul netulburat, iar cel care a pierdut poate încerca din nou după eliberarea magistralei. Câștigă, în mod natural, maestrul care emite adresa cu valoarea numerică mai mică, pentru că zero logic domină pe magistrală.

Al doilea mecanism este *întinderea ceasului* (*clock stretching*) și rezolvă problema sclavului lent. Ceasul este generat de maestru, dar linia SCL este tot cu colector deschis, deci un sclav care are nevoie de timp poate să o țină trasă la masă după ce maestrul a eliberat-o.

Maestrul, care verifică nivelul real al liniei înainte de a considera perioada de ceas încheiată, constată că SCL este încă jos și așteaptă. În felul acesta, un convertor analog-numeric care are nevoie de câteva sute de microsecunde pentru o conversie, sau o memorie EEPROM care trebuie să își încheie ciclul de scriere, poate încetini temporar magistrala fără să piardă date. Este un mecanism de control al fluxului pe care UART nu îl are și pe care SPI, așa cum vom vedea, de asemenea nu îl are.

9.5 Magistrala SPI: principiul de funcționare

Interfața SPI, de la *Serial Peripheral Interface*, pornește de la premise opuse. Nu încearcă să economisească fire, ci să transfere date cât mai repede și cu cât mai puțină logică. Rezultatul este un protocol atât de simplu încât, la rigoare, nici nu este un protocol: este un mecanism hardware de schimb de biți, peste care fiecare periferic își definește propriile convenții. De aceea SPI nu are, spre deosebire de I²C, o specificație oficială unică; există un mod de a face lucrurile, respectat de toată lumea, cu mici variații.

Comunicația este serială și sincronă, de tip maestru-sclav, și este *full-duplex*, adică datele circulă în ambele sensuri simultan, pe fire diferite. Maestrul, de regulă microcontrolerul, generează semnalul de ceas și controlează integral desfășurarea transferului; sclavii, perifericele, nu fac decât să reacționeze.

9.5.1 Cele patru linii

Magistrala folosește patru linii de semnal, ale căror denumiri descriu direct funcția fiecăreia.

SCLK (*Serial Clock*), notată în documentația Atmel și SCK, este linia de ceas. Este generată exclusiv de maestru și ritmează toate transferurile. Frecvența ei nu trebuie să fie constantă și nici stabilă; important este doar ca fronturile să respecte timpii minimi ceruți de sclav. Aceasta este o consecință directă a faptului că ceasul este transmis: sclavul nu are nevoie de niciun oscilator propriu pentru a comunica.

MOSI (*Master Out, Slave In*) transportă datele de la maestru la sclav. Este o ieșire pentru maestru și o intrare pentru fiecare sclav.

MISO (*Master In, Slave Out*) transportă datele în sens invers, de la sclav la maestru. Este o intrare pentru maestru și o ieșire pentru sclav. Faptul că cele două sensuri au fire separate este cel care face comunicația full-duplex posibilă și, așa cum vom vedea, chiar obligatorie.

$\overline{SS}$ (*Slave Select*), numită la unele periferice și CS (*Chip Select*), este linia de selecție. Bara de deasupra numelui arată că linia este activă în zero logic: sclavul este selectat când linia este trasă jos și ignorat când este lăsată sus. Un sclav neselectat nu ascultă ceasul și, foarte important, își trece ieșirea MISO în stare de înaltă impedanță, adică se deconectează electric de la linie. Fără această deconectare, mai mulți sclavi legați la același fir MISO s-ar bate pentru controlul lui.

9.5.2 Registrul de deplasare circular

Ideea centrală a interfeței SPI, și lucrul care trebuie înțeles înainte de orice altceva, este că transferul nu este o transmisie, ci un *schimb*. Atât maestrul, cât și sclavul conțin câte un registru de deplasare pe opt biți, iar cele două registre sunt legate cap la coadă prin firele MOSI și MISO, formând un singur registru circular pe șaisprezece biți. Semnalul de ceas comandă deplasarea în amândouă simultan.

La fiecare impuls de ceas se întâmplă exact două lucruri, în același timp: bitul de la un capăt al registrului maestrului iese pe MOSI și intră în celălalt capăt al registrului sclavului, iar bitul de la capătul registrului sclavului iese pe MISO și intră în registrul maestrului. Un bit iese și un bit intră, la fiecare dispozitiv, la fiecare front activ. După opt impulsuri de ceas, tot conținutul registrului maestrului a ajuns în registrul sclavului și tot conținutul registrului sclavului a ajuns în registrul maestrului: cele două dispozitive au schimbat între ele câte un octet. Figura 9.3 ilustrează această buclă.

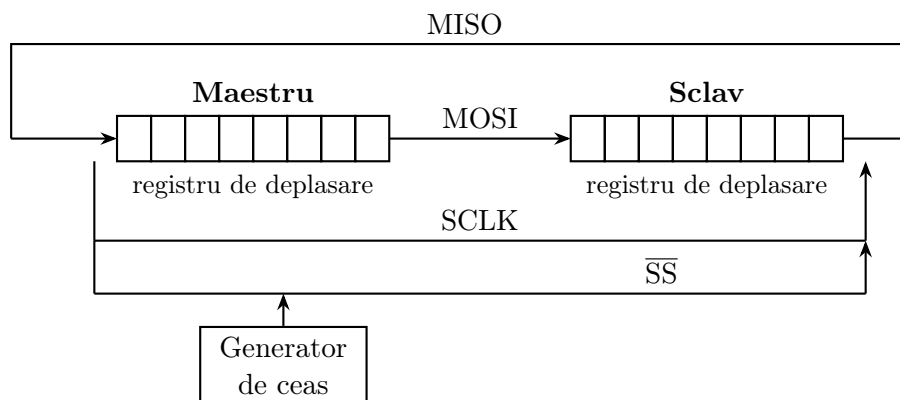


Figura 9.3 Registrele de deplasare ale maestrului și sclavului formează un inel închis prin liniile MOSI și MISO. La fiecare impuls de ceas, din fiecare registru iese un bit și intră un bit, astfel încât după opt impulsuri cele două dispozitive au schimbat între ele câte un octet.

Consecința acestui mecanism este surprinzătoare la prima vedere, dar inevitabilă: **pe SPI nu se poate citi fără a scrie și nu se poate scrie fără a citi**. Nu există o operație de „numai transmisie” și nici una de „numai recepție”, pentru că deplasarea în cele două registre este comandată de același ceas. Dacă maestrul vrea doar să trimită un octet, primește oricum în schimb octetul aflat în registrul sclavului, pe care de regulă îl ignoră. Dacă vrea doar să citească, trebuie totuși să trimită ceva, un octet oarecare, numit adesea *octet fictiv* (*dummy byte*), pentru simplul motiv că altfel nu ar avea de ce să genereze impulsuri de ceas, iar fără impulsuri de ceas sclavul nu are cum să deplaseze nimic. Această simetrie obligatorie explică forma tuturor funcțiilor de bibliotecă SPI, care nu au o funcție de scriere și una de citire, ci o singură funcție de transfer care primește un octet și întoarce un octet.

O a doua consecință este că sclavul nu poate răspunde imediat. Octetul pe care sclavul îl trimite în timpul unui transfer este cel pe care îl avea deja pregătit în registru la începutul transferului, nu unul calculat pe baza octetului primit în același timp. Dialogul obișnuit cu un periferic SPI are, de aceea, două etape: în primul transfer maestrul trimite o comandă

și primește un octet fără valoare, iar în al doilea transfer trimite un octet fictiv și primește răspunsul la comanda anterioară. Cine ignoră acest decalaj citește sistematic răspunsul cu o poziție întârziere.

9.6 Transferul datelor și modurile SPI

Cu mecanismul lămurit, desfășurarea concretă a unui transfer devine ușor de descris. Maestrul aduce mai întâi linia $\overline{SS}$ a sclavului dorit în zero logic. Din acest moment, sclavul respectiv este activ: își conectează ieșirea MISO la magistrală și își pregătește registrul de deplasare. Maestrul înscrie apoi octetul de trimis în registrul său de date, ceea ce declanșează automat generarea celor opt impulsuri de ceas. Pe măsură ce impulsurile se succed, biții pleacă pe MOSI și, simultan, biții sclavului sosesc pe MISO. La al optulea impuls, transferul s-a încheiat; maestrul citește din registrul său de date octetul primit de la sclav. Dacă mai are octeți de transferat, repetă operația fără a atinge linia de selecție. La sfârșitul întregii tranzacții, readuce $\overline{SS}$ în unu logic, ceea ce încheie dialogul cu sclavul și îl obligă pe acesta să își deconecteze ieșirea.

Linia de selecție are, deci, două roluri distincte. Primul, evident, este acela de a alege sclavul. Al doilea, mai puțin evident dar la fel de important, este acela de a delimita tranzacția: multe periferice își resetează logica internă la fiecare front descrescător al liniei de selecție, iar comenzile lor sunt interpretate relativ la începutul tranzacției. O memorie flash, de exemplu, așteaptă un octet de comandă urmat de trei octeți de adresă; dacă selecția este dezactivată la mijlocul acestei secvențe, comanda este abandonată în întregime.

9.6.1 Polaritatea și faza ceasului

Până acum am spus că biții se deplasează „la fiecare impuls de ceas”, fără a preciza care front al impulsului face ce anume. Aici apare singurul loc în care SPI cere o configurare atentă, și tot aici se află sursa celei mai frecvente erori de punere în funcțiune.

Un impuls de ceas are două fronturi, unul crescător și unul descrescător. Un registru de deplasare are nevoie de amândouă, dar pentru operații diferite: pe un front, ieșirea este actualizată, adică următorul bit este scos pe linie; pe celălalt front, intrarea este eșantionată, adică bitul prezent pe linie este capturat. Ordinea celor două operații nu poate fi arbitrară. Bitul trebuie mai întâi scos pe linie și lăsat să se stabilizeze, și abia apoi citit de partener; dacă ambele s-ar face pe același front, receptorul ar citi linia exact în momentul în care aceasta comută, adică ar citi o valoare nedefinită. Cele două operații se fac, prin urmare, pe fronturi opuse, la o jumătate de perioadă distanță, ceea ce oferă marginea de timp necesară.

Rămâne de stabilit *care* front face eșantionarea, iar răspunsul este dat de doi parametri de configurare.

Polaritatea ceasului, notată CPOL, spune în ce stare se află linia SCLK atunci când nu se transferă nimic. Cu CPOL egal cu zero, ceasul stă în zero logic, iar primul front al fiecărui impuls este crescător. Cu CPOL egal cu unu, ceasul stă în unu logic, iar primul front este descrescător.

Faza ceasului, notată CPHA, spune care dintre cele două fronturi ale impulsului face

eșantionarea. Cu CPHA egal cu zero, eșantionarea se face pe primul front al impulsului, iar schimbarea datelor pe al doilea. Cu CPHA egal cu unu, lucrurile se inversează: datele se schimbă pe primul front și se eșantionează pe al doilea.

Combinând cele două valori binare rezultă patru configurații, numite prin tradiție modurile 0, 1, 2 și 3, așa cum arată Tabelul 9.1.

Tabela 9.1 Cele patru moduri de funcționare SPI

Mod	CPOL	CPHA	Ceasul în re-paus	Eșantionare	Schimbarea datelor
0	0	0	jos	front crescător (primul)	front descrescător
1	0	1	jos	front descrescător (al doilea)	front crescător
2	1	0	sus	front descrescător (primul)	front crescător
3	1	1	sus	front crescător (al doilea)	front descrescător

Modurile 0 și 3 sunt cele mai răspândite; foarte multe periferice le acceptă pe amândouă, pentru că în ambele eșantionarea se face pe frontul crescător, iar diferența este doar starea de repaus a liniei. Modul 0 este configurația implicită a bibliotecii Arduino și a majorității memoriilor, cardurilor SD și convertoarelor.

9.6.2 De ce un mod greșit nu produce niciun mesaj de eroare

Merită insistat asupra a ceea ce se întâmplă când maestrul și sclavul sunt configurate în moduri diferite, pentru că este o situație pe care studentul o va întâlni cu siguranță în laborator.

Să presupunem că maestrul lucrează în modul 0, deci scoate biții pe frontul descrescător și îi citește pe cel crescător, iar sclavul este în modul 1, deci face invers. La primul front crescător, maestrul eșantionează linia MOSI a lui însuși - de fapt sclavul eșantionează MOSI - dar sclavul nu eșantionează încă nimic, pentru că el așteaptă frontul descrescător. Când acesta vine, maestrul a scos deja bitul următor pe linie. Sclavul capturează, așadar, bitul următor în loc de cel curent, iar toată succesiunea este deplasată cu o jumătate de perioadă. În unele combinații, bitul este eșantionat exact în momentul comutării, iar rezultatul depinde de întârzierile parazite ale traseului, ceea ce face ca erorile să apară intermitent și să pară aleatoare.

Ce nu se întâmplă, în schimb, este orice fel de semnalizare. Magistrala SPI nu are bit de paritate, nu are sumă de control, nu are confirmare de tipul ACK și nu are, de fapt, niciun mecanism prin care un dispozitiv să spună celuilalt că ceva nu este în regulă. Registrul de deplasare al maestrului se va umple cu opt biți la fel de conștiincios ca și cum totul ar fi corect, iar funcția de transfer va întoarce o valoare perfect plauzibilă. Simptomul este că un senzor raportează valori absurde sau constante, că un afișaj arată pixeli împrăștiati, că un

card SD nu se inițializează. Diagnosticul cere fie consultarea foii de catalog pentru modul cerut de periferic, fie un osciloscop cu care să se compare fronturile, fie, în lipsa acestora, încercarea metodică a celor patru moduri. Un mod SPI greșit este eroarea silențioasă prin excelență, iar aceasta este prețul plătit pentru simplitatea protocolului.

Observație

Aceeași categorie de erori silențioase include și ordinea biților. Majoritatea perifericelor așteaptă bitul cel mai semnificativ primul, dar unele lucrează invers, iar hardware-ul AVR permite ambele variante prin bitul DORD din registrul de control. Dacă ordinea este greșită, se recepționează exact aceiași opt biți, dar în oglindă; valoarea 0x01 devine 0x80, iar programul nu are cum să observe diferența.

9.7 Topologii SPI

Modul în care se leagă mai mulți sclavi la același maestru admite câteva variante, iar alegerea între ele este, în esență, un compromis între numărul de pini consumați și libertatea de a adresa individual fiecare dispozitiv.

Cazul cel mai simplu este acela al unui *singur sclav*. Cele patru linii merg direct de la maestru la periferic și nu sunt partajate cu nimeni. Într-o astfel de configurație, linia de selecție poate fi uneori legată permanent la masă, dacă perifericul acceptă acest lucru; multe însă nu îl acceptă, tocmai pentru că folosesc frontul liniei de selecție ca reper de sincronizare internă.

Configurația obișnuită cu mai multe periferice este cea *în paralel*, numită și cu selecție independentă. Toți sclavii împart liniile SCLK, MOSI și MISO, dar fiecare primește o linie $\overline{SS}$ proprie, comandată de un pin distinct al microcontrolerului. La un moment dat, maestrul activează o singură linie de selecție; sclavul astfel ales participă la transfer, iar ceilalți își țin ieșirile MISO în înaltă impedanță și ignoră ceasul, chiar dacă îl primesc. Avantajele sunt independența completă a dispozitivelor, care pot avea viteze și moduri SPI diferite, și simplitatea logică. Costul este în pini: pentru n sclavi sunt necesari $3 + n$ pini. Cu opt periferice se ajunge la unsprezece pini, adică peste jumătate din resursele unui ATmega328P, iar avantajul față de legăturile dedicate se erodează.

A treia variantă este *înlănțuirea (daisy chain)*. Aici sclavii sunt legați în serie: ieșirea MISO a primului merge la intrarea MOSI a celui de al doilea, ieșirea celui de al doilea la intrarea celui de al treilea și așa mai departe, iar ieșirea ultimului se întoarce la maestru. Liniile SCLK și $\overline{SS}$ sunt comune tuturor. Registrele de deplasare ale sclavilor formează astfel un singur registru lung, iar dacă sunt trei dispozitive cu câte opt biți, maestrul trebuie să trimită douăzeci și patru de biți, care se vor așeza, la sfârșitul transferului, câte opt în fiecare dispozitiv. Ridicarea liniei de selecție comune validează simultan datele în toate.

Câștigul este evident: numărul de pini rămâne patru indiferent câți sclavi sunt înlănțuiți. Prețul este la fel de evident: dispozitivele nu mai pot fi adresate individual, orice actualizare cere rescrierea întregului lanț, iar durata unui transfer crește proporțional cu numărul de dispozitive. Metoda se folosește acolo unde toate dispozitivele sunt de același fel și se actu-

alizează oricum împreună; exemplul clasic este un șir de drivere de afișaj cu șapte segmente sau de registre de ieșire care comandă un panou de LED-uri. Nu se folosește niciodată pentru senzori diferiți, care trebuie interogați separat. Trebuie adăugat că înlanțuirea cere ca perifericele să o suporte explicit, adică să transmită mai departe pe ieșire biții pe care i-au primit; nu orice circuit SPI face asta.

9.8 Modulul SPI al microcontrolerului ATmega328P

Microcontrolerul ATmega328P conține un modul SPI hardware complet, legat la patru pini ai portului B: PB2 pentru $\overline{SS}$, PB3 pentru MOSI, PB4 pentru MISO și PB5 pentru SCK. Pe placa Arduino Uno aceștia sunt pinii numerotați 10, 11, 12 și 13. Modulul este controlat prin trei registre.

SPCR (*SPI Control Register*) conține biții de configurare. Bitul SPE activează modulul; până când acesta nu este pus pe unu, cei patru pini rămân intrări-ieșiri obișnuite. Bitul MSTR selectează rolul: unu pentru maestru, zero pentru sclav. Biții CPOL și CPHA stabilesc modul de funcționare descris anterior. Bitul DORD alege ordinea biților, iar SPIE activează întreruperea de sfârșit de transfer. În fine, biții SPR1 și SPR0 selectează factorul de divizare al ceasului: combinațiile 00, 01, 10 și 11 dau, respectiv, $f_{osc}/4$, $f_{osc}/16$, $f_{osc}/64$ și $f_{osc}/128$.

SPSR (*SPI Status Register*) conține indicatorii de stare. Cel important este SPIF, indicatorul de transfer încheiat: hardware-ul îl pune pe unu atunci când cele opt impulsuri de ceas s-au consumat. Bitul WCOL semnalează o coliziune de scriere, adică o încercare de a înscrie în registrul de date în timp ce un transfer este în curs. Bitul SPI2X, singurul care se poate scrie, dublează frecvența ceasului față de valoarea selectată prin SPR1 și SPR0, permițând atingerea frecvenței maxime de $f_{osc}/2$.

SPDR (*SPI Data Register*) este registrul de date și, în realitate, fața vizibilă a registrului de deplasare. Scrierea în el declanșează un transfer, iar citirea din el întoarce octetul primit de la partener. Aceasta este traducerea directă, în termeni de registre, a ideii de schimb: o singură adresă de memorie servește și la trimitere, și la primire.

Exemplu

Un ATmega328P funcționează la 16 MHz și se dorește comunicația cu un periferic care acceptă cel mult 2 MHz pe linia de ceas. Ce configurație se alege și cât durează transferul unui octet?

Rezolvare. Factorii de divizare disponibili fără SPI2X sunt 4, 16, 64 și 128, care dau frecvențele 4 MHz, 1 MHz, 250 kHz și 125 kHz. Prima valoare depășește limita perifericului, deci se alege divizarea cu 16, adică SPR0 pus pe unu și SPR1 pe zero, rezultând un ceas de 1 MHz. Perioada unui bit este de 1 μ s, deci cei opt biți ai unui octet se transferă în 8 μ s. La aceasta se adaugă timpul consumat de program pentru scrierea în SPDR, pentru testarea indicatorului SPIF și pentru comanda liniei de selecție, care la 16 MHz înseamnă câteva zecimi de microsecundă pe instrucțiune. Se observă deja o proprietate importantă: la viteze mari de ceas, cheltuiala de program devine comparabilă cu durata

transferului propriu-zis, iar rata utilă efectivă rămâne sensibil sub cea teoretică.

9.8.1 Transferul unui octet prin acces direct la registre

Codul de mai jos implementează, la nivelul registrelor, inițializarea unui maestru SPI și transferul unui octet. Este exact secvența din care pornesc și bibliotecile de nivel înalt.

Exemplu 9.1 Inițializarea modulului SPI în regim de maestru

```

1 void SPI_MasterInit(void)
2 {
3     // MOSI (PB3), SCK (PB5) si SS (PB2) sunt iesiri
4     DDRB |= (1 << PB3) | (1 << PB5) | (1 << PB2);
5     // MISO (PB4) este intrare
6     DDRB &= ~(1 << PB4);
7     // Activare SPI, regim de maestru, ceas fosc/16
8     SPCR = (1 << SPE) | (1 << MSTR) | (1 << SPR0);
9     // Linia de selectie inactiva (High) intre transferuri
10    PORTB |= (1 << PB2);
11 }
```

Ordinea operațiilor nu este întâmplătoare. Direcțiile pinilor se stabilesc înaintea activării modulului, pentru că hardware-ul SPI preia controlul asupra lor abia după ce bitul SPE devine unu, dar respectă direcțiile impuse prin DDRB. Pinul $\overline{SS}$ merită o atenție specială: dacă este configurat ca intrare și cineva îl trage în zero logic, modulul comută automat din regim de maestru în regim de sclav, ștergând bitul MSTR. Este un mecanism prevăzut pentru sistemele cu mai mulți maestri, dar într-o aplicație obișnuită produce o defecțiune greu de explicat: comunicația se oprește brusc și fără motiv aparent. De aceea, pinul PB2 se configurează întotdeauna ca ieșire, chiar dacă selecția sclavului se face pe alt pin.

Exemplu 9.2 Transferul unui octet și citirea răspunsului

```

1 uint8_t SPI_MasterTransfer(uint8_t cData)
2 {
3     // Selectarea sclavului: SS pe Low
4     PORTB &= ~(1 << PB2);
5     // Scrierea în SPDR porneste transferul
6     SPDR = cData;
7     // Așteptare până la terminarea celor 8 impulsuri de ceas
8     while (!(SPSR & (1 << SPIF))) { }
9     // Deselectarea sclavului
10    PORTB |= (1 << PB2);
11    // Citirea din SPDR întoarce octetul primit de la sclav
12    return SPDR;
13 }
```

Bucula de așteptare este o *interogare ciclică (polling)*: procesorul verifică în mod repetat indicatorul SPIF până când acesta devine unu. Pentru un transfer de câteva microsecunde, soluția este perfect acceptabilă. Dacă transferurile sunt lungi, se poate activa în schimb întreruperea prin bitul SPIE și se poate trata sfârșitul transferului în rutina ISR(SPI_STC_vect), lăsând procesorul liber între timp. Se observă, de asemenea, că citirea registrului SPDR șterge automat indicatorul SPIF, ceea ce pregătește transferul următor.

Partea de sclav este simetrică și chiar mai simplă, pentru că sclavul nu are nimic de configurat în privința ceasului: el îl primește.

Exemplu 9.3 Configurarea modului SPI în regim de sclav și recepția

```

1 void SPI_SlaveInit(void)
2 {
3     // MOSI (PB3), SCK (PB5) și SS (PB2) sunt intrări
4     DDRB &= ~(1 << PB3) | (1 << PB5) | (1 << PB2);
5     // MISO (PB4) este ieșire
6     DDRB |= (1 << PB4);
7     // Activare SPI în regim de sclav (MSTR = 0)
8     SPCR = (1 << SPE);
9 }
10
11 uint8_t SPI_SlaveReceive(void)
12 {
13     // Așteptare până la sosirea unui octet complet
14     while (!(SPSR & (1 << SPIF))) { }
15     return SPDR;
16 }
```

Aici apare cu claritate limitarea discutată anterior. Sclavul întoarce octetul primit, dar ceea ce el a trimis maestrului în același timp este conținutul pe care SPDR îl avea înainte de transfer. Dacă sclavul vrea să răspundă la o comandă, trebuie să înscrie răspunsul în SPDR imediat după recepția comenzii, pentru ca el să fie deplasat afară în transferul următor.

9.9 Aplicații cu bibliotecile SPI și Wire

Mediul Arduino oferă pentru fiecare dintre cele două magistrale câte o bibliotecă standard, care ascunde manipularea registrelor și, mai ales, tratează corect detaliile ușor de greșit. Folosirea lor este recomandată în aplicații; înțelegerea registrelor rămâne însă necesară pentru a ști ce fac aceste biblioteci și de ce anume eșuează atunci când eșuează.

9.9.1 Biblioteca SPI

Biblioteca SPI pune la dispoziție noțiunea de *tranzacție*, prin care se declară viteza, ordinea biților și modul cerut de periferic. Declarația este necesară pentru că într-un program pot exista mai multe periferice SPI cu cerințe diferite, iar biblioteca reconfigurează hardware-

ul la începutul fiecărei tranzacții.

Exemplu 9.4 Citirea unui registru de la un periferic SPI

```
1  #include <SPI.h>
2
3  const uint8_t PIN_SS = 10;
4
5  void setup()
6  {
7      Serial.begin(9600);
8      pinMode(PIN_SS, OUTPUT);
9      digitalWrite(PIN_SS, HIGH);    // periferic deselectat
10     SPI.begin();
11 }
12
13 uint8_t citesteRegistru(uint8_t adresa)
14 {
15     uint8_t valoare;
16
17     // 1 MHz, bitul cel mai semnificativ primul, modul 0
18     SPI.beginTransaction(SPISettings(1000000, MSBFIRST, SPI_MODE0)
19         ↪ );
19     digitalWrite(PIN_SS, LOW);
20
21     // Primul transfer: se trimite comanda, se primește un octet
22     ↪ fara valoare
23     SPI.transfer(adresa | 0x80);    // bitul 7 marchează o citire
24     // Al doilea transfer: octet fictiv, se primește răspunsul
25     valoare = SPI.transfer(0x00);
26
27     digitalWrite(PIN_SS, HIGH);
28     SPI.endTransaction();
29
30     return valoare;
31 }
32
33 void loop()
34 {
35     Serial.println(citesteRegistru(0x0F), HEX);
36     delay(500);
37 }
```

Programul ilustrează toate ideile discutate. Selecția este comandată explicit de program, nu de bibliotecă, pentru că numărul pinului depinde de montaj. Cele două transferuri succesive reflectă decalajul inerent al registrului circular: primul poartă comanda, al doilea aduce răspunsul, iar octetul fictiv 0x00 este trimis doar pentru a genera ceasul. Bitul cel mai semnificativ al adresei este pus pe unu pentru a marca o operație de citire; aceasta nu este o regulă a magistralei SPI, ci o convenție folosită de o mare parte dintre perifericele comerciale, pe care foaia lor de catalog o descrie.

9.9.2 Biblioteca Wire

Pentru I²C, biblioteca se numește **Wire** și implementează tranzacțiile descrise anterior. Programul de mai jos scrie o valoare într-un registru și apoi o citește înapoi, urmând exact secvențele descrise: scriere, apoi scriere urmată de start repetat și citire.

Exemplu 9.5 Scrierea și citirea unui registru pe magistrala I²C

```

1  #include <Wire.h>
2
3  const uint8_t ADRESA = 0x68;    // adresa pe 7 biti a perifericului
4
5  void setup()
6  {
7      Serial.begin(9600);
8      Wire.begin();               // porneste magistrala in regim de
      ↪ maestru
9      Wire.setClock(400000); // 400 kHz, daca perifericul suporta
10 }
11
12 // Scrie o valoare intr-un registru; intoarce 0 in caz de succes
13 uint8_t scrieRegistru(uint8_t reg, uint8_t valoare)
14 {
15     Wire.beginTransmission(ADRESA); // pregateste START + adresa
      ↪ + W
16     Wire.write(reg);                // primul octet: numarul
      ↪ registrului
17     Wire.write(valoare);            // al doilea octet: valoarea
18     return Wire.endTransmission(); // emite totul si genereaza
      ↪ STOP
19 }
20
21 uint8_t citesteRegistru(uint8_t reg)
22 {
23     Wire.beginTransmission(ADRESA);

```

```

24     Wire.write(reg);
25     Wire.endTransmission(false);    // false = START repetat,
        ↪ fara STOP
26
27     Wire.requestFrom(ADRESA, (uint8_t)1); // citește un octet
28     if (Wire.available()) {
29         return Wire.read();
30     }
31     return 0xFF;
32 }
33
34 void loop()
35 {
36     scrieRegistru(0x07, 0x1F);
37     Serial.println(citeșteRegistru(0x07), HEX);
38     delay(1000);
39 }

```

Câteva detalii merită comentate. Funcția `beginTransmission` nu trimite nimic pe magistrală; ea doar reține adresa, iar octeții adăugați cu `write` se acumulează într-un buffer intern. Abia `endTransmission` generează efectiv condiția de start, emite adresa și octeții și încheie cu o condiție de stop. Valoarea întoarsă de această funcție este codul de eroare și trebuie verificată: zero înseamnă succes, doi înseamnă NACK la adresă, adică periferic absent, iar trei înseamnă NACK la un octet de date. Argumentul `false` transmis lui `endTransmission` suprimă condiția de stop, astfel încât apelul următor să genereze un start repetat, exact așa cum cere tranzacția de citire descrisă mai devreme.

9.9.3 Scanarea magistralei I²C

Instrumentul cel mai util în lucrul cu această magistrală decurge direct din mecanismul de confirmare. Dacă maestrul emite o condiție de start urmată de un octet de adresă și apoi, indiferent de răspuns, o condiție de stop, el nu transferă nicio dată, dar află un lucru esențial: dacă a primit ACK, la acea adresă există un dispozitiv; dacă a primit NACK, nu există. Repetând operația pentru toate adresele valide se obține lista completă a perifericelor prezente.

Exemplu 9.6 Scanarea magistralei I2C pentru gasirea adreselor

```

1  #include <Wire.h>
2
3  void setup()
4  {
5      Serial.begin(9600);
6      while (!Serial) { }

```

```

7      Wire.begin();
8      Serial.println("Scanare magistrala I2C...");
9  }
10
11 void loop()
12 {
13     uint8_t gasite = 0;
14
15     for (uint8_t adresa = 8; adresa < 120; adresa++) {
16         Wire.beginTransmission(adresa);
17         uint8_t stare = Wire.endTransmission(); // START + adresa
18             ↳ + STOP
19
19         if (stare == 0) { // ACK: dispozitiv prezent
20             Serial.print("Dispozitiv gasit la adresa 0x");
21             if (adresa < 16) {
22                 Serial.print("0");
23             }
24             Serial.println(adresa, HEX);
25             gasite++;
26         }
27     }
28
29     if (gasite == 0) {
30         Serial.println("Niciun dispozitiv. Verificati alimentarea,
31             ↳ ");
32         Serial.println("masa comuna si rezistentele de ridicare.")
33             ↳ ;
34     }
35
36     delay(5000);
37 }

```

Intervalul de adrese parcurs, de la 8 la 119, evită cele două grupe rezervate de la capetele domeniului. Un rezultat gol nu înseamnă neapărat că perifericul este defect; cele mai frecvente cauze sunt inversarea liniilor SDA și SCL, lipsa masei comune între module și absența rezistoarelor de ridicare. Un rezultat în care apar dispozitive la toate adresele indică, de obicei, o linie SDA blocată în zero logic, situație în care fiecare interogare pare confirmată.

Exemplu

Se cere configurarea magistralei I²C hardware a unui ATmega328P la 16 MHz pentru o frecvență a ceasului de 100 kHz și apoi pentru 400 kHz. Modulul TWI generează frecvența după relația

$$f_{SCL} = \frac{f_{osc}}{16 + 2 \cdot TWBR \cdot 4^{TWPS}},$$

unde TWBR este registrul de rată, iar TWPS selectează un divizor suplimentar.

Rezolvare. Se alege mai întâi cel mai mic divizor suplimentar, adică TWPS egal cu zero, ceea ce face factorul 4^{TWPS} egal cu unu. Relația devine $f_{SCL} = f_{osc}/(16 + 2 \cdot TWBR)$. Pentru 100 kHz, raportul f_{osc}/f_{SCL} este $16\,000\,000/100\,000 = 160$, deci $16 + 2 \cdot TWBR = 160$, de unde $TWBR = 72$. Pentru 400 kHz, raportul este 40, deci $16 + 2 \cdot TWBR = 40$ și $TWBR = 12$. Ambele valori sunt întregi și încap comod într-un registru de opt biți, deci ambele frecvențe se obțin exact. Se observă că, spre deosebire de UART, unde eroarea ratei de transfer este o problemă permanentă, aici o abatere de la valoarea nominală nu ar avea nicio consecință funcțională: sclavii urmăresc ceasul primit, oricare ar fi el, cu condiția să nu depășească limita lor maximă.

9.10 Comparație între UART, SPI și I²C

Cele trei interfețe seriale studiate în acest capitol și în cel precedent nu se înlocuiesc una pe alta, ci acoperă situații diferite. Tabelul 9.2 rezumă deosebirile care contează la alegerea uneia dintre ele.

Tabela 9.2 Comparație între cele trei interfețe seriale

Criteriu	UART	SPI	I ² C
Fire de semnal	2 (TxD, RxD)	3 comune plus câte o linie de selecție	2 (SDA, SCL)
Ceas comun	nu, fiecare capăt are ceasul lui	da, generat de maestru	da, generat de maestru
Viteză uzuală	9600 până la 115200 bps	1 până la 10 Mbit/s	100 kHz, 400 kHz, mai rar peste
Număr de dispozitive	2, punct la punct	1 maestru și n sclavi, cu $3 + n$ pini	până la 112 adrese pe magistrală
Sens simultan	full-duplex	full-duplex	half-duplex
Confirmare	niciuna, doar paritate opțională	niciuna la nivel de protocol	ACK sau NACK după fiecare octet
Complexitate	mică, dar cere rate potrivite	mică, registre de deplasare	mai mare, colector deschis și arbitraj
Utilizare tipică	legături cu calculatorul, module GSM, depanare	memorii, carduri SD, afișaje, convertoare rapide	senzori lenți, ceasuri de timp real, EEPROM

Regula practică de alegere se poate formula în câteva propoziții. Dacă dispozitivul de la celălalt capăt este departe, este un calculator sau un modul care vorbește oricum serial asincron, se folosește UART. Dacă transferul este mare și rapid, iar numărul de periferice este mic, se folosește SPI: un afișaj grafic, un card de memorie sau un convertor analog-numeric care eșantionează de mii de ori pe secundă nu au altă opțiune rezonabilă. Dacă perifericele sunt multe, mici și lente, iar pini sunt puțini, se folosește I²C: un senzor de temperatură care se citește o dată pe secundă nu are nevoie de un megabit pe secundă, iar economia de pini este decisivă.

Trebuie observat, în fine, că cele trei interfețe coexistă fără probleme pe același microcontroller, pentru că folosesc module hardware și pini diferiți. Un proiect obișnuit are simultan o legătură UART către calculator pentru depanare, o magistrală I²C cu doi sau trei senzori și o legătură SPI către un card de memorie sau către un afișaj. Ceea ce diferă este doar cine ocupă procesorul și pentru cât timp, iar acesta este subiectul capitolelor următoare.

9.11 Concluzii

Acest capitol a pornit de la limitele comunicației asincrone: absența unui ceas comun, care obligă la o potrivire strânsă a ratelor de transfer și plafonează viteza, și imposibilitatea de a lega mai mult de două dispozitive, care face ca numărul de pini să devină rapid factorul limitativ. Ambele magistrale sincrone prezentate rezolvă aceste probleme transportând explicit un semnal de ceas, alături de date.

Magistrala SPI face acest lucru cu patru linii și aproape fără protocol. Mecanismul ei fundamental este registrul de deplasare circular: maestrul și sclavul își leagă registrele cap la coadă prin MOSI și MISO, iar la fiecare impuls de ceas un bit iese și un bit intră, astfel încât citirea și scrierea se petrec obligatoriu simultan. Selecția dispozitivelor se face prin linii dedicate, ceea ce este simplu, dar costă un pin de fiecare sclav; înlanțuirea evită acest cost cu prețul pierderii adresării individuale. Cele patru moduri, definite de polaritatea și faza ceasului, trebuie potrivite exact între maestru și sclav, iar o nepotrivire produce date greșite fără niciun fel de semnalizare.

Magistrala I²C face același lucru cu doar două linii, plătind pentru asta cu un protocol considerabil mai elaborat. Colectorul deschis și rezistoarele de ridicare fac magistrala partajabilă fără conflicte electrice și stau la baza tuturor mecanismelor ei: condițiile de start și de stop, care folosesc singurele tranziții interzise în timpul datelor, confirmarea ACK sau NACK după fiecare octet, arbitrajul între maștri și întinderea ceasului de către un sclav lent. Adresarea pe șapte biți înlocuiește firul de selecție, permițând peste o sută de dispozitive pe aceleași două fire.

Alegerea între ele nu este o chestiune de preferință, ci de cerințe: viteză și volum de date pentru SPI, economie de pini și număr mare de periferice lente pentru I²C, distanță și compatibilitate cu lumea din afară pentru UART.

Întrebări de verificare

1. Explicați de ce o legătură UART nu poate lega trei dispozitive pe aceleași două fire, în timp ce o magistrală I²C poate lega zece. Ce element al protocolului face diferența și ce înlocuiește el la nivel fizic?
2. De ce transportul explicit al semnalului de ceas elimină problema abaterii oscilatoarelor? Arătați ce anume din mecanismul de eșantionare al receptorului asincron devine inutil într-o interfață sincronă.
3. Descrieți ce se întâmplă electric pe linia SDA dacă un dispozitiv ar avea o ieșire obișnuită, în contratimp, în loc de una cu colector deschis, iar un al doilea dispozitiv ar trage linia la masă în același moment. De ce rezistorul de ridicare rezolvă situația?
4. O magistrală I²C are o capacitate totală estimată la 200 pF și rezistoare de ridicare de 10 k Ω . Estimați constanta de timp a frontului crescător și comparați-o cu durata unei perioade de ceas la 400 kHz. Ce concluzie trageți despre valoarea aleasă a rezistoarelor?
5. Explicați de ce condiția de start nu poate fi confundată cu un bit de date oarecare. Ce regulă a protocolului garantează acest lucru și în ce moment al perioadei de ceas se schimbă în mod normal linia SDA?
6. Un maestru citește cinci octeți consecutivi de la un sclav. Precizați, pentru fiecare dintre cei cinci octeți, cine generează bitul de confirmare și ce valoare trebuie să aibă. Ce s-ar întâmpla dacă maestrul ar confirma și ultimul octet?
7. Justificați afirmația că pe magistrala SPI nu se poate citi fără a scrie. Pornind de la structura registrului de deplasare circular, arătați de ce este necesar un octet fictiv atunci când maestrul dorește doar să primească date.
8. Un periferic SPI cere modul 2, iar programul îl configurează pe modul 0. Descrieți ce se întâmplă la nivelul fronturilor de ceas și explicați de ce programul nu primește niciun mesaj de eroare. Cum se poate identifica greșeala în lipsa unui osciloscop?
9. Comparați topologia SPI cu selecție independentă cu cea în lanț din punctul de vedere al numărului de pini și al posibilității de a adresa individual dispozitivele. Pentru care dintre următoarele aplicații ați alege înlanțuirea: opt afișaje cu șapte segmente identice, sau un card SD împreună cu un senzor de presiune? Justificați.
10. Pe un ATmega328P la 16 MHz se dorește un ceas SPI de cel mult 500 kHz. Determinați combinația biților SPR1 și SPR0 și frecvența efectiv obținută, apoi calculați durata transferului unui bloc de 512 octeți, fără a lua în considerare cheltuiala de program.
11. Explicați rolul pinului PB2 atunci când modulul SPI al lui ATmega328P lucrează în regim de maestru. De ce configurarea lui ca intrare poate face ca microcontrolerul să

treacă neașteptat în regim de sclav și cum se manifestă acest lucru în funcționarea programului?

12. Un program de scanare a magistralei I²C raportează dispozitive la toate adresele parcurse. Ce defect al montajului explică acest rezultat și de ce anume fiecare interogare pare confirmată? Cum ați verifica ipoteza cu un simplu multimetru?

10. Proiectarea și testarea aplicațiilor cu microcontrolere

Proiectare și testare
Un exemplu de referință
Descrierea sistemului și caietul de sarcini
Partiționarea sistemului
Alegerea microcontrolerului
Detalierea subsistemelor
Activitatea principală
Scrierea și testarea primară a codului
Planul de testare
Depanarea sistematică
Documentarea
Întrebări de verificare

10.1 Proiectare și testare

Capitolele anterioare au tratat, pe rând, părțile componente ale unui microcontroler: porturile de intrare-ieșire, întreruperile, modulele de tip timer, convertorul analog-numeric, interfețele seriale. Fiecare dintre acestea a fost prezentat izolat, cu registrele și cu regulile lui de funcționare. Un inginer care stăpânește toate aceste module nu știe însă, prin asta, să facă un produs. Între „am aprins un LED cu PWM” și „am livrat un sistem care funcționează nesupravegheat doi ani într-o seră” există o distanță care nu se acoperă cu cunoștințe suplimentare despre registre, ci cu *metodă*.

Experiența arată un lucru care surprinde la început: proiectele cu microcontrolere eșuează aproape întotdeauna din motive de metodă, nu de programare. Foarte rar un proiect cade pentru că cineva nu a înțeles cum se configurează un prescaler. Cade pentru că nimeni nu a scris limpede, la început, ce trebuie să facă sistemul; pentru că împărțirea pe module a fost făcută din mers, iar fiecare funcție a ajuns să atingă fiecare variabilă; pentru că testarea a fost lăsată la sfârșit, când costul unei corecții este deja de zeci de ori mai mare decât ar fi fost la început. Acest capitol este despre acele decizii care se iau înainte de a scrie prima linie de cod și despre activitatea, adesea nepopulară, care confirmă că ceea ce s-a scris chiar funcționează.

10.1.1 De ce eșuează un proiect

Prima cauză este *cerința neclară*. Beneficiarul spune „vreau ca sistemul să reacționeze repede” sau „vreau să fie sigur”. Nimeni nu contrazice asemenea afirmații, pentru că nimeni nu știe ce înseamnă. La recepție, beneficiarul constată că sistemul reacționează în 300 ms și afirmă că nu este „repede”, proiectantul răspunde că nici nu s-a cerut altceva, iar discuția nu are cum să se încheie, pentru că nu există un criteriu obiectiv la care ambele părți să se raporteze. O cerință care nu poate fi verificată printr-o măsurătoare nu este o cerință, ci o dorință.

A doua cauză este *delimitarea proastă a subsistemelor*. Când codul crește fără o structură decisă dinainte, funcțiile încep să comunice între ele prin variabile globale, fiecare modificând starea celuilalt. Un asemenea program funcționează, de obicei, până la prima modificare. Atunci se descoperă că schimbarea pragului de umiditate impune modificări în rutina care comandă releul, care la rândul ei strică afișarea, iar corecția afișării reintroduce eroarea inițială. Sistemul devine imposibil de întreținut nu pentru că ar fi complex, ci pentru că nu are granițe interne.

A treia cauză, cea mai costisitoare, este *testarea amânată*. Este tentant să scrii tot programul și abia apoi să îl încerci, pentru că testarea pe parcurs pare o pierdere de timp. În realitate, atunci când integrezi zece module netestate și rezultatul nu funcționează, nu ai nicio informație despre locul erorii: oricare dintre cele zece module, sau oricare dintre interacțiunile dintre ele, poate fi vinovat. Dacă în schimb fiecare modul a fost verificat separat, iar sistemul se strică la adăugarea celui de al zecelea, spațiul de căutare s-a redus la un singur modul și la interfața lui. Testarea incrementală nu este o precauție, ci singura metodă prin care depanarea rămâne o activitate finită.

Observație

Costul corectării unei erori crește puternic cu momentul în care este descoperită. O cerință scrisă greșit și corectată pe hârtie costă o oră. Aceeași cerință descoperită după ce a fost implementată costă rescrierea codului. Descoperită după ce s-a proiectat cablajul, costă un nou cablaj. Descoperită după ce o sută de plăci sunt montate în teren, costă deplasarea la o sută de locații. Nimic din ceea ce face un proiectant nu are un randament mai bun decât timpul petrecut clarificând cerințele înainte de a începe.

10.1.2 Cele două etape și ciclul dintre ele

Dezvoltarea unei aplicații cu microcontroler se desfășoară în două etape mari: *etapa de proiectare*, care începe cu descrierea sistemului și se încheie cu primul draft de cod, și *etapa de testare*, care începe cu simularea și se încheie cu documentarea. Ordonate, pașii sunt următorii: descrierea completă a sistemului; întocmirea caietului de sarcini; determinarea necesarului hardware și software; realizarea diagramelor bloc care partiționează sistemul; detalierea subsistemelor prin diagrame de activități; scrierea primului draft de cod pornind de la aceste diagrame; simularea în medii specializate; realizarea unui modul hardware simplu

pentru încercări; compilarea și execuția programului; testarea pe baza planului de testare; corectarea sistemului și, dacă este cazul, îmbogățirea planului de testare; în sfârșit, documentarea a tot ce s-a făcut.

Reprezentarea acestor pași ca o listă este însă înșelătoare, pentru că sugerează o curgere într-un singur sens. În realitate, între ultimii pași și primii există o buclă de reacție, arătată în Figura 10.1: dacă la testare sistemul nu se comportă conform caietului de sarcini, nu se corectează doar codul, ci se reia proiectarea de la nivelul la care s-a produs greșeala. Uneori se descoperă chiar că însuși caietul de sarcini era greșit, pentru că beneficiarul a formulat o cerință imposibilă sau contradictorie. Un proiect sănătos parcurge acest ciclu de mai multe ori, cu amplitudine din ce în ce mai mică.

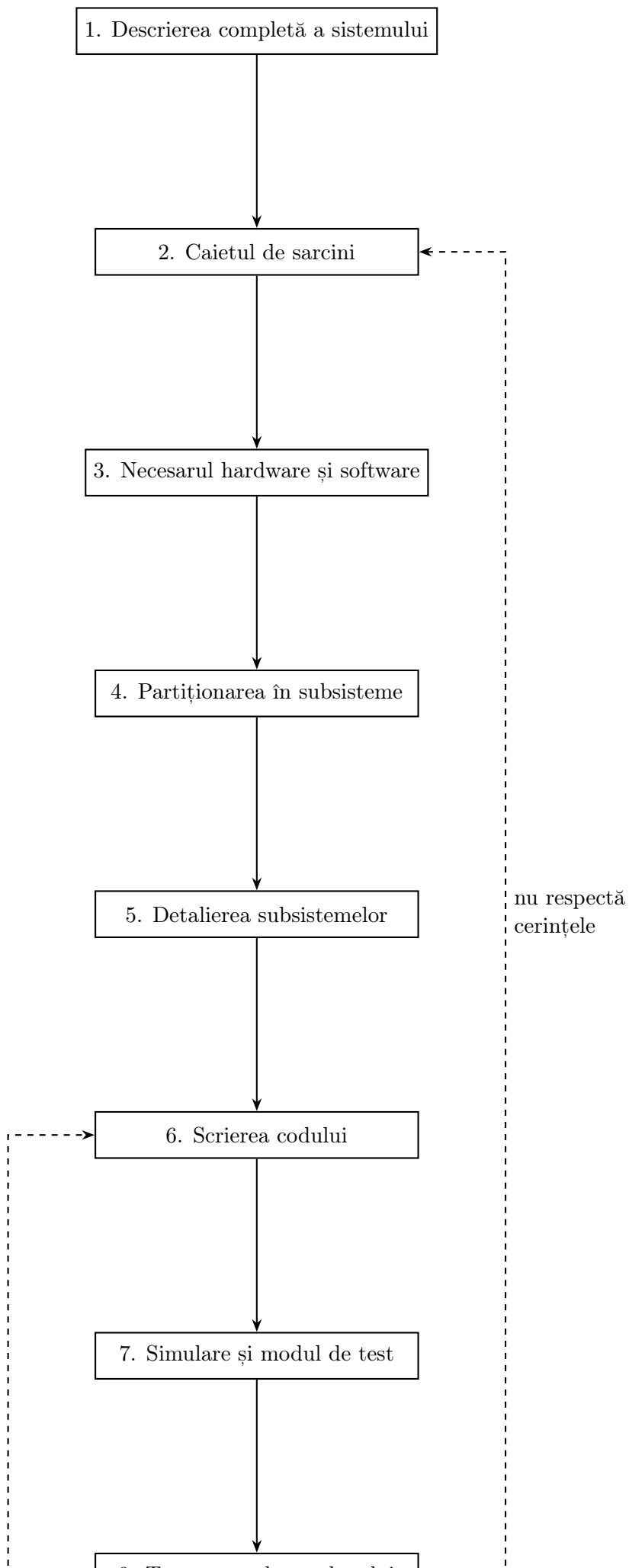
Trebuie subliniat un aspect adesea trecut cu vederea: etapa de proiectare produce documente, nu cod. Diagramele bloc, tabelele de cerințe, tabelele de tranziții și harta pinilor sunt rezultate în sine, care rămân în urma proiectului și îi determină valoarea pe termen lung. Un sistem care funcționează, dar despre care nu există niciun document, este un sistem pe care nimeni nu îl mai poate modifica, adică un sistem cu durată de viață egală cu memoria autorului lui.

10.2 Un exemplu de referință

Pentru ca discuția să nu rămână abstractă, tot capitolul va urmări un singur exemplu, parcurs de la cerință până la planul de testare: un *sistem de udare automată* pentru o seră mică. Sistemul măsoară umiditatea solului, iar când aceasta scade sub un prag deschide o electrovalvă printr-un releu, pentru un interval limitat, după care lasă apa să se distribuie în sol înainte de a măsura din nou. Are un buton de udare manuală, un buton de oprire, două LED-uri de stare și transmite pe interfața serială un jurnal al evenimentelor.

Situația de plecare este cea tipică în practica inginerescă și seamănă cu exemplul discutat la curs pentru un ansamblu de porți automate: o firmă beneficiară trimite un document care descrie, în limbaj obișnuit, cum ar trebui să funcționeze instalația. Urmează mai multe întâlniri în care principiul de funcționare se schițează împreună, se decid componentele de putere, se stabilește cine ce face. Este esențial să se clarifice de la început *granița de responsabilitate*: dacă partea de forță, adică electrovalva, releul și sursa, este realizată de altcineva, atunci obiectul contractului este algoritmul de comandă și interfața electrică dintre cele două părți, iar acea interfață trebuie descrisă în scris, cu nivele logice, curenți și polarități, nu convenită verbal.

Din prima discuție rezultă întotdeauna un al doilea document, la fel de important ca descrierea primită: *lista de întrebări*. Ea conține tot ce descrierea nu spune. Ce se întâmplă dacă senzorul de umiditate se defectează și indică permanent sol uscat? Se udă la nesfârșit? Ce se întâmplă dacă cineva apasă simultan butonul de udare manuală și pe cel de oprire? Ce se întâmplă dacă alimentarea cade în timpul udării și revine după cinci secunde: se reia udarea de la zero, sau sistemul consideră că a udat deja? Un proiectant experimentat se recunoaște după calitatea acestei liste, pentru că fiecare întrebare pusă acum este o eroare care nu va mai fi descoperită în teren.



10.3 Descrierea sistemului și caietul de sarcini

Descrierea sistemului este documentul beneficiarului, scris în limbajul lui. *Caietul de sarcini* este documentul comun, scris în limbaj tehnic, care transformă descrierea într-un set de afirmații pe care le poți verifica una câte una. Trecerea de la primul la al doilea este pasul de proiectare cu cel mai mare efect asupra reușitei, și în același timp singurul pas pe care studenții sunt tentați să îl sară complet.

10.3.1 Ce înseamnă o cerință verificabilă

O cerință este verificabilă dacă, având sistemul în față și un aparat de măsură, două persoane independente ajung obligatoriu la aceeași concluzie despre îndeplinirea ei. Comparăm două formulări ale aceleiași intenții:

Prima: „sistemul trebuie să răspundă repede la depășirea pragului”. A doua: „sistemul trebuie să comande releul în cel mult 50 ms de la momentul în care mărimea măsurată depășește pragul configurat”. Prima formulare nu spune ce înseamnă „repede”, nu spune de unde se măsoară, nu spune ce se întâmplă la prag. A doua fixează un moment de start (depășirea pragului), un moment de final (comanda releului), o valoare-limită (50 ms) și implicit o metodă de verificare: se aduce intrarea peste prag cu un front cunoscut, se privește cu osciloscopul intrarea și comanda releului și se măsoară intervalul dintre ele. Diferența dintre cele două nu este de stil, ci de natură: a doua poate fi infirmată printr-o măsurătoare, prima nu.

O cerință bine scrisă are, de regulă, patru elemente: un subiect (ce parte a sistemului), o acțiune observabilă, o condiție de declanșare și o valoare numerică cu unitate de măsură și toleranță. Cuvintele care trebuie să dea de gândit atunci când apar într-un caiet de sarcini sunt „repede”, „stabil”, „sigur”, „prietenos”, „aproximativ”, „dacă este posibil”. Fiecare dintre ele ascunde o negociere care nu a avut loc.

10.3.2 Cerințe funcționale și cerințe nefuncționale

Cerințele funcționale descriu ce face sistemul: ce citește, ce calculează, ce comandă, în ce ordine, în cât timp. Ele sunt cele la care se gândește oricine și sunt relativ ușor de formulat, pentru că decurg direct din descrierea beneficiarului.

Cerințele nefuncționale descriu în ce condiții trebuie să facă acele lucruri, iar ele sunt sursa majorității surprizelor târzii. Consumul mediu de curent determină dacă sistemul poate fi alimentat de la un panou solar mic sau are nevoie de rețea. Domeniul de temperatură de lucru elimină din start componente și decide dacă oscilatorul poate fi cel intern, cu deriva lui termică, sau trebuie cuarț. Costul unitar decide dacă microcontrolerul are 32 KB de memorie de program sau 8 KB. Comportarea la alimentare instabilă decide dacă sistemul are nevoie de un circuit de supraveghere a tensiunii și de un numărător de gardă. Durata de viață și cerințele de mentenanță decid dacă parametrii de configurare stau în memoria EEPROM, modificabili din teren, sau sunt constante compilate în program. Niciuna dintre aceste cerințe nu apare în descrierea beneficiarului, dar toate schimbă proiectul dacă sunt

descoperite târziu.

Tabelul 10.1 arată un fragment din caietul de sarcini al sistemului de udare. Se observă că fiecare cerință are un cod, ceea ce permite trimiterea la ea din planul de testare și din raportul de încercări, și o coloană care spune cum se verifică. O cerință pentru care nu se poate completa a treia coloană trebuie rescrisă.

Tabela 10.1 Fragment din caietul de sarcini al sistemului de udare automată. Cerințele CF sunt funcționale, cele CN nefuncționale.

Cod	Enunțul cerinței	Mod de verificare
CF1	Sistemul măsoară umiditatea solu- lui o dată la $1 \pm 0,1$ s.	Cronometrarea impulsu- rilor pe pinul de test cu osciloscopul, 100 de ci- cluri.
CF2	Dacă media ultimelor 8 măsurători scade sub pragul configurat, sis- temul deschide electrovalva în cel mult 200 ms.	Injectarea unei tensiuni sub prag, măsurarea în- târzierii cu osciloscopul.
CF3	Durata unei udări nu depășește 60 s, indiferent de indicația senzo- rului.	Blocarea senzorului la valoarea de sol uscat, cronometrarea ieșirii.
CF4	După o udare, sistemul nu redes- chide valva timp de minimum 30 de minute.	Test funcțional cu bază de timp accelerată.
CF5	La apăsarea butonului de oprire, valva se închide în cel mult 50 ms, din orice stare.	Măsurare cu osciloscopul pe cele două semnale.
CF6	Dacă tensiunea de la senzor iese din domeniul 0,3...4,7 V, sistemul intră în starea de avarie și nu co- mandă valva.	Deconectarea senzorului, apoi scurtcircuitarea lui la masă.
CN1	Consumul mediu al plăcii de co- mandă, cu valva închisă, sub 20 mA.	Măsurare cu amperme- tru pe durata a 10 mi- nute.
CN2	Funcționare corectă între -5 și $+55$ °C.	Încercare în incintă cli- matică, la ambele ex- treme.
CN3	O întrerupere a alimentării nu lasă valva deschisă.	Întreruperea alimentării în timpul udării, de 20 de ori.
CN4	Parametrii de prag și durată sunt modificabili fără reprogramare.	Verificarea salvării în EEPROM și a citirii după resetare.

Merită observat cum cerințele CF3, CF6 și CN3 răspund exact întrebărilor din lista întocmită la prima întâlnire. Aceasta este funcția listei de întrebări: fiecare întrebare pusă devine, după clarificare, o cerință scrisă, iar sistemul capătă un comportament definit inclusiv

în situațiile pe care beneficiarul nu le-a avut în minte.

Exemplu

Verificați dacă cerința CF5 (închiderea valvei în cel mult 50 ms de la apăsarea butonului de oprire) este realizabilă cu o arhitectură în care butoanele sunt citite prin scrutare, la fiecare 2 ms, iar o apăsare este confirmată după 20 ms de nivel stabil. Releul folosit are, conform catalogului, un timp de acționare de 10 ms.

Rezolvare. Bugetul de timp se construiește adunând, în cel mai defavorabil caz, toate întârzierile de pe lanțul de la cauză la efect. Apăsarea poate cădea imediat după o citire, deci se pierde până la un interval de scrutare, adică 2 ms. Urmează confirmarea antirebound, 20 ms. Decizia din bucla principală și scrierea în registrul portului consumă sub 1 ms, deci considerăm 1 ms. Releul are nevoie de 10 ms pentru ca armătura să comute. Total: $2 + 20 + 1 + 10 = 33$ ms, cu o rezervă de 17 ms față de limita de 50 ms. Cerința este realizabilă, iar rezerva este suficientă pentru a absorbi variațiile de execuție ale buclei. Dacă însă confirmarea antirebound ar fi fost aleasă la 50 ms, cum se întâmplă adesea fără justificare, cerința ar fi fost încălcată din proiectare, iar eroarea ar fi apărut abia la măsurătorile de recepție.

10.4 Partiționarea sistemului

Având caietul de sarcini, urmează întrebarea din ce părți este alcătuit sistemul. Partiționarea înseamnă împărțirea ansamblului în *subsisteme*, fiecare cu o responsabilitate unică, și stabilirea modului în care acestea comunică între ele. Instrumentul obișnuit este diagrama bloc, care arată ierarhia subsistemelor, interacțiunea dintre partea software și partea hardware și interfețele către module externe.

10.4.1 Interfața se stabilește înaintea implementării

Regula fundamentală a partiționării este că *interfața unui subsistem se fixează înainte ca subsistemul să fie implementat*. Motivul este practic: atât timp cât interfața nu este fixată, niciun alt subsistem nu poate fi scris, pentru că nu se știe ce va primi și ce va returna. Din momentul în care interfața este scrisă, fie și doar ca prototipuri de funcții cu comentarii, toate celelalte module pot fi dezvoltate în paralel, iar cel care implementează subsistemul are libertate deplină înăuntru, atât timp cât respectă contractul.

Consecința este că interfața trebuie descrisă complet: ce intră, ce iese, în ce unități de măsură, în ce domeniu de valori, cu ce eroare, ce se întâmplă la valori invalide, cât durează operația și dacă poate fi apelată dintr-o rutină de întrerupere. O interfață care spune doar `uint16_t citesteSenzor(void);` nu este completă; una care spune că funcția returnează umiditatea în procente $\times 10$, în domeniul 0 la 1000, valoarea 0xFFFF semnalând senzor invalid, că durează sub 200 μ s și că nu poate fi apelată din întrerupere, este.

Un al doilea criteriu de partiționare bună este ca fiecare subsistem să aibă un singur motiv de a se schimba. Dacă înlocuirea senzorului de umiditate cu altul, cu altă caracteristică, obligă la modificarea codului care aprinde LED-urile, partiționarea este greșită. Subsistemul

de achiziție trebuie să ascundă complet tipul senzorului în spatele unei valori exprimate în unități fizice, astfel încât restul programului să nu știe niciodată dacă senzorul este rezistiv sau capacitiv.

10.4.2 Diagrama bloc a exemplului

Sistemul de udare se împarte în mod natural în subsistemele din Figura 10.2. Subsistemul de achiziție citește canalul analogic, mediază eșantioanele și convertește în procente de umiditate. Subsistemul de interfață cu operatorul citește butoanele, le filtrează de rebound și comandă LED-urile. Subsistemul de comandă a valvei traduce o cerere logică într-o stare a pinului de releu și impune limitele de siguranță. Subsistemul de bază de timp, construit pe un timer, produce evenimentele periodice. Subsistemul de jurnalizare trimite pe interfața serială înregistrări ale evenimentelor. În mijloc stă aplicația propriu-zisă, care nu atinge niciun registru, ci ia doar decizii pe baza valorilor primite.

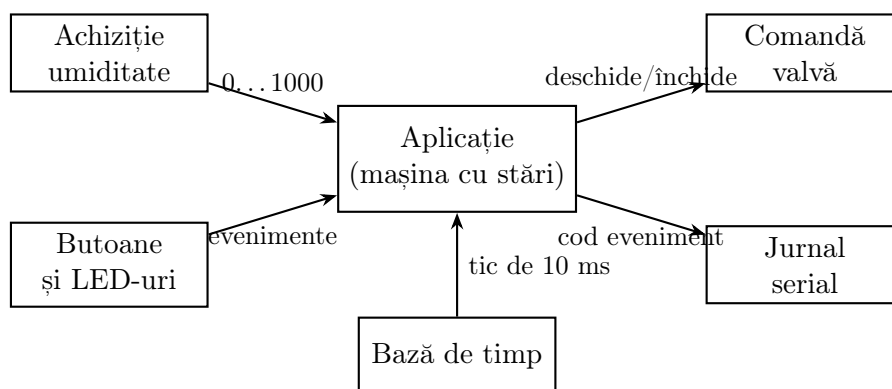


Figura 10.2 Partiționarea sistemului de udare pe subsisteme, cu indicarea mărimilor care trec prin fiecare interfață.

Partiționarea se continuă până la nivelul funcțional unic, adică până acolo unde fiecare bloc corespunde unei singure operații pe care o poți scrie ca funcție: inițializarea portului de intrare, citirea unui canal ADC, medierea, compararea cu pragul. Pentru fiecare bloc se decide în acest moment și *modul de implementare*: software sau hardware. Antireboundul butoanelor se poate face în software, prin confirmare temporală, sau în hardware, printr-un grup RC și un trigger Schmitt. Generarea unei baze de timp se poate face prin bucle de așteptare sau printr-un timer. Fiecare alegere are un preț, iar decizia se ia comparând costul componentelor cu costul timpului de procesor și cu riscul.

Observație

Un bloc funcțional bine delimitat se recunoaște după faptul că poate fi testat singur. Dacă pentru a verifica funcția de decizie a udării trebuie să conectezi senzorul real, sursa de 12 V și electrovalva, atunci funcția de decizie nu este un bloc separat, ci s-a amestecat cu accesul la periferice. Testabilitatea nu este o consecință a unei bune partiționări, ci criteriul ei.

10.5 Alegerea microcontrolerului

Alegerea microcontrolerului nu se face după obișnuință sau după ce piesă se găsește în sertar, ci ca o consecință a caietului de sarcini și a partiționării. După ce se știe ce semnale intră și ce semnale ies, numărul de pini necesari este determinat, iar după ce se știu perioadele de lucru, se determină perifericele necesare. Abia atunci se poate deschide catalogul.

Criteriile care contează sunt sistematizate în Tabelul 10.2, împreună cu ceea ce impune fiecare în cazul exemplului nostru. Se observă că niciunul nu este singur decisiv: alegerea finală este întotdeauna un compromis între cerințe tehnice, cost și disponibilitate.

Tabela 10.2 Criterii de alegere a microcontrolerului, aplicate sistemului de udare automată.

Criteriu		Ce impune aplicația	Consecința
Numărul de pini	de	1 intrare analogică, 3 intrări digitale, 3 ieșiri digitale, 2 pini pentru serial, 2 pentru programare	Un capsulă de 28 de pini este suficientă, cu rezervă
Periferice		Un canal ADC, un timer pentru baza de timp, un USART pentru jurnal	Nu sunt necesare SPI sau interfață I2C
Memoria de program	de	Cod de câteva mii de instrucțiuni, fără biblioteci grafice	8 KB ar ajunge; se alege 32 KB pentru dezvoltări ulterioare
Memoria de date	de	Tampon de mediere, tampon serial, variabile de stare	Câteva sute de octeți de SRAM
Memoria ne-volatilă	ne-	Praguri și durate modificabile din teren (CN4)	Este obligatorie o memorie EEPROM
Viteza		Decizii la 10 ms, fără calcul intensiv	Câțiva MHz sunt suficienți; frecvența mai mică reduce consumul
Consumul		Sub 20 mA în medie (CN1)	Sunt necesare moduri de consum redus și oprirea perifericelor nefolosite
Domeniul de temperatură	de	$-5 \dots +55$ °C (CN2)	Variantă industrială sau verificarea derivei oscilatorului intern
Disponibilitate și preț		Serie mică, livrare rapidă, unelte accesibile	Familie larg răspândită, cu lanț de compilare gratuit

Pentru exemplul de față, un ATmega328P satisface toate liniile tabelului cu rezervă: cei 23 de pini de intrare-ieșire acoperă larg necesarul de nouă pini, convertorul analog-numeric are șase canale multiplexate, cele trei module de tip timer lasă unul liber pentru baza de timp și unul pentru eventuala comandă în PWM, cei 32 KB de memorie flash și 2 KB de SRAM sunt cu mult peste ce cere aplicația, iar memoria EEPROM de 1 KB rezolvă cerința

de configurare din teren. Este important de reținut că această concluzie a rezultat din tabel, nu a precedat tabelul: dacă aplicația ar fi cerut, de exemplu, opt canale analogice și un consum mediu sub $100\ \mu\text{A}$, aceeași metodă ar fi condus la altă piesă.

Un criteriu care nu apare în tabel, dar care în practică decide adesea, este *rezerva*. Un proiect care folosește toți pinii, toată memoria și tot timpul de procesor este un proiect fără viitor: prima cerință suplimentară a beneficiarului obligă la schimbarea microcontrolerului și la reproiectarea cablajului. Regula practică este de a păstra o rezervă de cel puțin douăzeci la sută la pini, memorie și timp de execuție.

10.6 Detalierea subsistemelor

Cu subsistemele definite și cu interfețele fixate, urmează descrierea a ceea ce face fiecare înăuntru. Pentru aceasta se folosește un limbaj de modelare standardizat, cel mai răspândit fiind *UML* (Unified Modeling Language), din care aplicațiilor cu microcontrolere le sunt utile în special *diagrama de activități* și *diagrama de stări*. Prima descrie o succesiune de operații și decizii, cu un nod de start, noduri de acțiune, noduri de decizie, transferuri de control și un nod de sfârșit. Criteriul de calitate al unei asemenea diagrame este simplu și sever: din ea trebuie să se poată scrie codul direct, fără a mai lua nicio decizie de proiectare.

10.6.1 Bucla principală față de arhitectura condusă de evenimente

Există două organizări fundamentale ale unui program de microcontroler. În prima, *bucla de scrutare* (polling), programul parcurge la nesfârșit aceeași secvență: citește toate intrările, calculează, actualizează toate ieșirile. Avantajul este simplitatea și faptul că momentul fiecărei operații este previzibil. Dezavantajul este că timpul de reacție la un eveniment este, în cel mai rău caz, egal cu durata întregii bucle, iar dacă o singură operație este lentă, de exemplu o transmisie serială care așteaptă golirea registrului, toate celelalte suferă.

În a doua organizare, *arhitectura condusă de evenimente*, perifericele semnalează prin întreruperi momentul în care s-a întâmplat ceva, iar rutinele de întrerupere depun evenimentul într-o coadă sau ridică un fanion, pe care bucla principală îl consumă. Reacția este rapidă și independentă de durata buclei, dar apar probleme noi: partajarea variabilelor între rutina de întrerupere și programul principal, necesitatea calficatorului *volatile*, secțiunile critice care trebuie protejate. În practică, aproape orice aplicație reală este un hibrid: întreruperile captează evenimentele rapide și marchează timpul, iar bucla principală, ritmată de un tic produs de un timer, ia deciziile.

10.6.2 Mașina cu stări finite

Forma naturală în care se organizează logica unei aplicații încorporate este *mașina cu stări finite*. Ideea este ca sistemul să se afle, în orice moment, într-una dintr-un număr mic de stări bine definite, iar comportamentul lui să depindă nu doar de intrări, ci și de starea curentă. Aceeași apăsare de buton înseamnă altceva când sistemul udă și altceva când sistemul așteaptă, iar mașina cu stări exprimă acest lucru direct, în loc să îl ascundă într-un labirint de instrucțiuni condiționale imbricate.

Avantajul practic este că mașina cu stări face imposibile, prin construcție, combinațiile nedorite. Dacă tranziția din starea de pauză obligatorie către starea de udare nu există în tabel, atunci nu există nici în cod, deci sistemul nu poate uda de două ori la rând, indiferent ce apasă utilizatorul sau ce indică un senzor defect. Cerințele de siguranță se traduc astfel în absența unor tranziții, formă mult mai ușor de verificat decât o condiție complicată.

Descrierea completă a unei mașini cu stări are două forme echivalente: diagrama, utilă pentru discuție, și *tabelul de tranziții*, util pentru implementare și pentru testare. Figura 10.3 arată diagrama sistemului de udare, iar Tabelul 10.3 tabelul corespunzător. Se recomandă ca tabelul să fie complet, adică pentru fiecare stare să fie enumerate toate evenimentele posibile, inclusiv cele care nu produc nicio tranziție; altfel rămân colțuri nedefinite ale comportamentului, iar acele colțuri sunt exact locurile în care apar defectele.

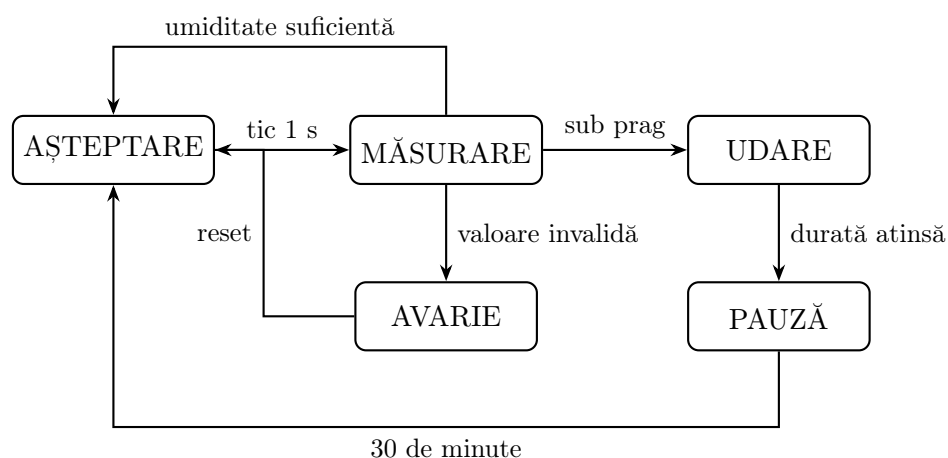


Figura 10.3 Mașina cu stări a sistemului de udare. Absența unei tranziții directe din PAUZĂ în UDARE face imposibilă, prin construcție, udarea repetată.

Traducerea tabelului în cod este mecanică, ceea ce este chiar scopul lui. Starea se păstrează într-o variabilă de tip enumerare, iar bucla principală conține o instrucțiune de selecție cu un caz pentru fiecare stare. Regula de aur este ca niciun alt loc din program să nu modifice variabila de stare: dacă starea se schimbă din trei locuri diferite, avantajul mașinii cu stări dispare complet.

Exemplu 10.1 Structura mașinii cu stări, derivată direct din tabelul de tranziții

```

1 typedef enum {
2     ST_AȘTEPTARE = 0,
3     ST_MASURARE,
4     ST_UDARE,
5     ST_PAUZA,
6     ST_AVARIE
7 } stare_t;
8
9 static stare_t stareCurenta = ST_AȘTEPTARE;

```

Tabela 10.3 Tabelul de tranziții al sistemului de udare automată.

Stare rentă	cu- Eveniment condiție	sau	Acțiune	Stare urmă- toare
AȘTEPTARE	Tic de o secundă		Pornește conver- sia ADC	MĂSURARE
AȘTEPTARE	Buton udare manu- ală		Jurnalizează cere- rea	UDARE
AȘTEPTARE	Buton oprire		Nicio acțiune	AȘTEPTARE
MĂSURARE	Media < prag		Deschide valva, pornește crono- metrul	UDARE
MĂSURARE	Media $\geq$ prag		Jurnalizează va- loarea	AȘTEPTARE
MĂSURARE	Valoare în afara do- meniului		Aprinde LED-ul de avarie	AVARIE
UDARE	Durata maximă atinsă		Închide valva, pornește tempo- rizarea	PAUZĂ
UDARE	Buton oprire		Închide valva, jur- nalizează	PAUZĂ
UDARE	Valoare invalidă		Închide valva, semnalizează	AVARIE
PAUZĂ	Temporizare expi- rată		Stinge LED-ul de udare	AȘTEPTARE
PAUZĂ	Buton udare manu- ală		Ignoră, jurnali- zează refuzul	PAUZĂ
AVARIE	Resetare de la opera- tor		Șterge indicația de avarie	AȘTEPTARE
AVARIE	Orice alt eveniment		Nicio acțiune	AVARIE

```

10
11 void ruleazaMasinaDeStari(event_t ev)
12 {
13     switch (stareCurenta) {
14
15     case ST_AȘTEPTARE:
16         if (ev == EV_TIC_1S) {
17             adcPorneșteConversia();
18             stareCurenta = ST_MASURARE;
19         } else if (ev == EV_BUTON_MANUAL) {
20             valvaDeschide();
21             cronometruPornește(DURATA_MAX_S);
22             stareCurenta = ST_UDARE;

```

```
23     }
24     break;
25
26     case ST_MASURARE:
27         if (ev == EV_VALOARE_INVALIDA) {
28             ledAvarie(1);
29             stareCurenta = ST_AVARIE;
30         } else if (ev == EV_SUB_PRAG) {
31             valvaDeschide();
32             cronometruPornire(DURATA_MAX_S);
33             stareCurenta = ST_UDARE;
34         } else if (ev == EV_PESTE_PRAG) {
35             stareCurenta = ST_ASTEPTARE;
36         }
37         break;
38
39     case ST_UDARE:
40         if (ev == EV_DURATA_EXPIRATA || ev == EV_BUTON_OPRIRE) {
41             valvaInchide();
42             cronometruPornire(PAUZA_MIN_S);
43             stareCurenta = ST_PAUZA;
44         } else if (ev == EV_VALOARE_INVALIDA) {
45             valvaInchide();
46             ledAvarie(1);
47             stareCurenta = ST_AVARIE;
48         }
49         break;
50
51     case ST_PAUZA:
52         if (ev == EV_PAUZA_EXPIRATA) {
53             stareCurenta = ST_ASTEPTARE;
54         }
55         /* cererile manuale sunt ignorate in mod deliberat */
56         break;
57
58     case ST_AVARIE:
59         if (ev == EV_RESET_OPERATOR) {
60             ledAvarie(0);
61             stareCurenta = ST_ASTEPTARE;
62         }
63         break;
```

```

64     }
65 }

```

10.7 Activitatea principală

Diagrama de activități a programului principal descrie ce se întâmplă de la alimentare până la oprire. Ea începe invariabil cu un bloc de inițializări: inițializarea variabilelor, inițializarea porturilor cu stabilirea direcțiilor, inițializarea modului de tip timer care va da baza de timp, inițializarea convertorului analog-numeric și a interfeței seriale, apoi validarea globală a întreruperilor. Ordinea nu este indiferentă. Pinul care comandă releul trebuie adus în starea inactivă *înainte* de a fi configurat ca ieșire, altfel, în intervalul dintre cele două instrucțiuni, ieșirea preia valoarea rămasă în registrul de date și valva se poate deschide pentru câteva microsecunde la fiecare pornire a sistemului.

După inițializări urmează bucla infinită. Structura ei tipică, în varianta ritmată de un tic, este arătată în Listingul de mai jos: bucla așteaptă apariția fanionului ridicat de rutina de întrerupere a timerului, colectează evenimentele de la subsisteme și le trimite mașinii cu stări. Se observă că bucla nu conține nicio întârziere activă și nicio operație lungă; tot ce durează mult, precum transmisia jurnalului, este fragmentat sau lăsat pe seama întreruperilor.

Exemplu 10.2 Bucla principala ritmata de un tic de 10 ms

```

1  volatile uint8_t bTic = 0;    /* ridicat de ISR, citit de bucla
    ↪ principala */
2
3  ISR(TIMER1_COMPA_vect)
4  {
5      bTic = 1;                 /* ISR foarte scurt: doar marcheaza
    ↪ timpul */
6  }
7
8  int main(void)
9  {
10     valvaInchide();            /* stare sigura INAINTE de a
    ↪ configura iesirea */
11     porturiInit();
12     timerInit();              /* intrerupere la fiecare 10 ms */
13     adcInit();
14     serialInit(9600);
15     parametriIncarcaDinEeprom();
16     sei();
17
18     for (;;) {
19         if (bTic) {

```

```

20         cli();
21         bTic = 0;           /* stergere protejata: ISR poate
                               ↪ scrie oricand */
22         sei();
23
24         eveniment_t ev = colecteazaEveniment();
25         ruleazaMasinaDeStari(ev);
26         cronometruTic();
27         jurnalTrimiteUnCaracter();
28     }
29 }
30 }

```

Citirea butoanelor merită o discuție separată, pentru că este locul în care apar cele mai multe erori de logică. Un buton nu se citește ca nivel, ci ca *tranzitie*: sistemul trebuie să reacționeze la momentul apăsării, nu la faptul că butonul este ținut apăsat. Metoda standard este de a păstra valoarea citită anterior a portului și de a o compara cu cea curentă; dacă diferă, s-a produs o schimbare, iar biții care s-au modificat arată care buton a fost acționat. Aceeași metodă rezolvă și problema apăsării simultane a mai multor butoane, pentru că valoarea întregului port se interpretează ca un cod, iar codurile nepermise se pot trata explicit.

Exemplu 10.3 Detectarea tranzitiilor pe portul de butoane, cu filtrare de rebound

```

1  static uint8_t vechiPINB = 0xFF;
2  static uint8_t stabilPINB = 0xFF;
3  static uint8_t nrTicuriStabile = 0;
4
5  eveniment_t citesteButoane(void)
6  {
7      uint8_t nouPINB = PINB & MASCA_BUTOANE;
8
9      if (nouPINB != stabilPINB) {           /* semnal inca instabil
                               ↪ */
10         stabilPINB = nouPINB;
11         nrTicuriStabile = 0;
12         return EV_NICIUNUL;
13     }
14
15     if (nrTicuriStabile < TICURI_ANTI_REBOUND) {
16         nrTicuriStabile++;
17         return EV_NICIUNUL;
18     }

```

```

19
20     if (stabilPINB == vechiPINB) {           /* nicio schimbare
        ↪ confirmata */
21         return EV_NICIUNUL;
22     }
23
24     uint8_t schimbate = stabilPINB ^ vechiPINB;
25     uint8_t apasate   = schimbate & ~stabilPINB; /* butoane
        ↪ active in zero */
26     vechiPINB = stabilPINB;
27
28     if (apasate & BIT_OPRIRE) return EV_BUTON_OPRIRE;
29     if (apasate & BIT_MANUAL)  return EV_BUTON_MANUAL;
30     if (apasate & BIT_RESET)   return EV_RESET_OPERATOR;
31     return EV_NICIUNUL;
32 }

```

Se observă prioritatea explicită dintre butoane: dacă utilizatorul apasă simultan oprirea și udarea manuală, sistemul returnează evenimentul de oprire. Aceasta nu este o coincidență de programare, ci o decizie de proiectare care trebuie să figureze în caietul de sarcini, pentru că are consecințe de siguranță. Un cod care lasă ordinea la voia întâmplării va funcționa diferit după orice reorganizare a instrucțiunilor.

10.8 Scrierea și testarea primară a codului

Cu diagramele finalizate, scrierea codului devine în bună măsură transcriere. Limbajul folosit este de regulă C sau C++, cu recurgere la limbaj de asamblare doar pentru secvențele în care contează fiecare ciclu de ceas sau accesul la instrucțiuni pe care compilatorul nu le generează. Compromisul este cunoscut: limbajul de asamblare oferă control complet asupra timpului de execuție și asupra dimensiunii codului, dar este legat de o familie de procesoare și este lent de scris și de întreținut; limbajul C oferă portabilitate și claritate, cu prețul unei cantități mici de cod suplimentar generat de compilator, care în aplicațiile obișnuite nu contează.

Regula esențială a acestei etape este să nu se scrie tot programul deodată. Codul se dezvoltă pe straturi, iar fiecare periferic se verifică separat, cu un program minimal, înainte de a fi integrat. Ordinea firească este de jos în sus: mai întâi se verifică faptul că microcontrolerul rulează la frecvența așteptată, ceea ce se face cel mai simplu comutând un pin într-o buclă și măsurând perioada cu osciloscopul; apoi se verifică portul de ieșire cu un LED; apoi intrarea cu un buton; apoi convertorul analog-numeric, alimentat de la un potențiomtru și cu rezultatul afișat pe interfața serială; apoi timerul, verificând cu osciloscopul perioada ticului; și abia la sfârșit se aduc împreună.

10.8.1 De ce nu se testează direct pe instalația reală

Nu este indicat ca primele încercări să se facă pe echipamentul real al beneficiarului. Motivul cel mai evident este că o eroare de software poate distruge hardware scump: o comandă simultană a două brațe ale unei punți în H scurtcircuitază sursa, o valvă lăsată deschisă inundă sera, un motor comandat cu sens greșit rupe un limitator mecanic. Al doilea motiv, mai puțin evident, este că instalația reală nu poate fi adusă la comandă în situațiile care contează: nu poți face solul uscat în cinci secunde ca să verifici pragul și nu poți repeta de o sută de ori o cădere de tensiune.

De aceea se construiește un *modul de simulare* ieftin, care imită comportamentul instalației la interfața electrică: intrările digitale se dau de la butoane sau de la un generator, intrările analogice de la potențiometre, ieșirile se observă pe LED-uri, iar semnalele rapide, precum PWM-ul, pe osciloscop. Pentru sistemul de udare, un asemenea modul înseamnă un potențiometru în locul senzorului de umiditate, trei butoane și trei LED-uri, dintre care unul în locul releului. Costul este de câțiva lei, iar câștigul este posibilitatea de a produce oricând orice combinație de intrări, inclusiv pe cele imposibile fizic, care sunt cele mai interesante pentru testare.

Alături de modulul fizic se folosesc și *simulatoare* software, care execută programul instrucțiune cu instrucțiune și permit inspectarea registrelor și a memoriei. Ele sunt de neînlocuit pentru verificarea logicii și pentru măsurarea numărului de cicluri consumate de o secvență, dar au o limitare pe care studentul trebuie să o știe: un simulator modelează microcontrolerul, nu lumea din jurul lui. Reboundul contactelor, zgomotul de pe linia analogică, căderea de tensiune la anclanșarea releului și cuplajul dintre cablul de forță și cel de semnal nu apar în simulator și sunt, în practică, sursa majorității defectelor.

10.8.2 Cod ușor de testat

Se poate scrie cod care se lasă testat și cod care nu se lasă. Diferența stă în separarea deciziei de acces la periferice. O funcție care citește singură registrul ADC, decide și scrie singură în portul releului nu poate fi verificată decât cu hardware. Aceeași logică, scrisă ca funcție pură care primește valoarea măsurată și pragurile și returnează decizia, poate fi compilată și rulată pe calculator, cu mii de combinații de intrări, în câteva secunde.

Exemplu 10.4 Funcție pură de decizie, verificabilă fără hardware

```

1  /* Returnează 1 dacă trebuie deschisă valva, 0 dacă nu.
2     umiditate: 0..1000 (procente x 10); prag: 0..1000;
3     histerezis: latimea zonei moarte, in aceleasi unitati;
4     udaAcum: starea curenta a valvei (1 deschisa). */
5  uint8_t decideUdarea(uint16_t umiditate, uint16_t prag,
6                      uint16_t histerezis, uint8_t udaAcum)
7  {
8      if (udaAcum) {
9          return (umiditate < prag + histerezis) ? 1 : 0;

```

```

10     }
11     return (umiditate < prag) ? 1 : 0;
12 }

```

Funcția de mai sus conține și o soluție la o problemă practică importantă. Fără *histerezis*, adică fără cele două praguri diferite pentru pornire și oprire, o valoare de umiditate care oscilează din cauza zgomotului în jurul pragului ar produce deschideri și închideri repetate ale releului, câteva pe secundă, cu uzura rapidă a contactelor și a electrovalvei. Zona moartă introdusă face ca, odată pornită udarea, oprirea să se producă numai după ce umiditatea a urcat vizibil peste prag. Aceeași problemă și aceeași soluție apar la orice termostat.

10.9 Planul de testare

După ce modulul de încercare este gata, se scrie *planul de testare*. Acesta nu este o listă de lucruri pe care le încerci când îți vine în minte, ci un document derivat din caietul de sarcini, în care fiecare cerință are cel puțin un test asociat și fiecare test are un criteriu de acceptare formulat numeric. Legătura se face prin codurile cerințelor, astfel încât la sfârșit să se poată demonstra că nicio cerință nu a rămas neverificată.

Testele trebuie construite pe trei direcții: să acopere toate cerințele din caietul de sarcini, să confirme criteriile de performanță în condiții reale, nu doar de laborator, și să verifice comportarea în scenarii accidentale, adică acolo unde nimeni nu a proiectat nimic anume.

10.9.1 Niveluri de testare

Testele unitare se aplică funcțiilor pur logice, care nu ating periferice: conversia din valoarea ADC în procente, medierea, decizia de udare, tabelul de tranziții. Ele se compilează pentru calculatorul gazdă și se rulează automat, ceea ce înseamnă că pot fi repetate la fiecare modificare a codului, cu un efort nul. Un test unitar bun include valorile de la limitele domeniului și valorile imediat vecine lor, pentru că acolo se ascund erorile de comparație.

Exemplu 10.5 Test unitar rulat pe calculator, fara microcontroler

```

1  #include <assert.h>
2
3  int main(void)
4  {
5      /* sub prag, valva inchisa: trebuie sa porneasca */
6      assert(decideUdarea(300, 400, 50, 0) == 1);
7      /* peste prag, valva inchisa: nu porneste */
8      assert(decideUdarea(500, 400, 50, 0) == 0);
9      /* exact pe prag: comparatia este stricta, deci nu porneste */
10     assert(decideUdarea(400, 400, 50, 0) == 0);
11     /* in zona de histerezis, valva deschisa: ramane deschisa */
12     assert(decideUdarea(430, 400, 50, 1) == 1);
13     /* peste prag plus histerezis: se inchide */

```

```
14     assert (decideUdarea(460, 400, 50, 1) == 0);  
15     return 0;  
16 }
```

Testele pe modul verifică un subsistem întreg, cu hardware simulat: se aplică la intrarea analogică o tensiune cunoscută de la un potențiomtru și se compară valoarea afișată pe interfața serială cu cea măsurată cu voltmetrul. Ele verifică nu doar logica, ci și configurarea perifericului: referința ADC, prescalarea, alinierea rezultatului.

Testele de sistem verifică ansamblul, prin scenarii care urmăresc folosirea reală: sol uscat, udare, oprire la timp, pauză, măsurare din nou. Aici se cronometrează și se compară cu limitele din caietul de sarcini.

Testele la limite exercitează valorile extreme: umiditate zero, umiditate maximă, exact valoarea pragului, exact durata maximă, contorul de timp în momentul depășirii capacității. Un contor de milisecunde pe 16 biți se rotește după aproximativ 65 de secunde, iar o comparație scrisă neatent va da un rezultat greșit exact atunci. Testul care surprinde acest defect trebuie să existe în plan, pentru că altfel defectul apare la beneficiar, o dată la câteva zeci de secunde sau o dată la 49 de zile, în funcție de lungimea contorului.

Testele de robustețe verifică ce se întâmplă în situații nedorite: deconectarea senzorului în timpul măsurării, scurtcircuitarea lui la masă, alimentarea instabilă, resetarea în timpul udării, apăsarea simultană a mai multor butoane, apăsarea butonului de o sută de ori pe secundă. Testarea atipică încearcă deliberat blocarea sistemului și combinații de intrări care nu au fost prevăzute la proiectare. Criteriul de acceptare nu este ca sistemul să facă ceva util, ci ca el să nu facă ceva periculos: să ajungă întotdeauna într-o stare sigură, cu valva închisă și cu avaria semnalizată.

Testele T11 și T12 nu au o cerință asociată, ceea ce este un semnal util: ele provin din testarea atipică și, dacă un asemenea test descoperă un defect, înseamnă că în caietul de sarcini lipsea o cerință. Corectarea nu se face doar în cod, ci și în caietul de sarcini, care se completează cu cerința nou descoperită. După orice corecție, testele se rulează din nou, integral și nu doar cel care a eșuat, pentru că o modificare făcută pentru a repara un defect este exact momentul în care se introduc altele. Practic, planul de testare crește pe măsură ce proiectul avansează, iar la sfârșit el este cea mai bună descriere a comportamentului real al sistemului.

Un ultim nivel, adesea prevăzut în contract, este trimiterea modelelor de simulare și a rezultatelor de testare către beneficiar, pentru ca acesta să efectueze propriile încercări. Este în interesul proiectantului ca acest lucru să se întâmple cât mai devreme, pentru că beneficiarul cunoaște instalația reală și va produce, fără să vrea, exact acele combinații de intrări la care proiectantul nu s-a gândit.

10.10 Depanarea sistematică

Oricât de bine ar fi organizat proiectul, vor exista defecte, iar modul în care sunt căutate face diferența între o oră și o săptămână. Depanarea nu este o activitate de inspirație, ci o

Tabela 10.4 Fragment din planul de testare al sistemului de udare automată.

Test	Cerință	Mod de efectuare	Criteriu de acceptare
T1	CF1	Se comută pinul de test la fiecare măsurare; se măsoară perioada	Perioada între 0,9 și 1,1 s pe 100 de cicluri
T2	CF2	Se coboară brusc tensiunea de la potențiometrul sub prag	Comanda releului apare în mai puțin de 200 ms
T3	CF3	Se menține potențiometrul la sol uscat	Valva se închide după cel mult 60 s
T4	CF4	După o udare, se cere udare manuală imediat	Cererea este refuzată și jurnalizată
T5	CF5	Se apasă oprirea în timpul udării, cu osciloscopul pe ambele semnale	Întârziere sub 50 ms, la 20 de repetări
T6	CF6	Se deconectează senzorul, apoi se leagă la masă	Trecere în AVARIE, valva rămâne închisă
T7	CF6	Se reconectează senzorul și se apasă resetarea	Revenire în AȘTEPTARE, jurnal corect
T8	CN1	Ampermetru în serie cu alimentarea, 10 minute	Curent mediu sub 20 mA
T9	CN3	Se întrerupe alimentarea în timpul udării	Valva închisă la revenire, sistem în stare sigură
T10	CN4	Se modifică pragul, se resetează sistemul	Valoarea nouă este activă după resetare
T11	-	Se apasă simultan toate butoanele, de 50 de ori	Sistemul rămâne în stare definitivă, fără blocaj
T12	-	Se comută alimentarea de 100 de ori la interval de o secundă	Nicio comandă parazită pe releu

procedură.

Primul pas este *reducerea la un caz minim reproductibil*. Un defect care apare „uneori, după vreo jumătate de oră” nu poate fi studiat. Trebuie găsită secvența cea mai scurtă de acțiuni care îl provoacă de fiecare dată, apoi eliminată, una câte una, orice parte a sistemului care nu este necesară pentru reproducere. Deconectăm afișajul: mai apare? Oprim jurnalul serial: mai apare? Înlocuim senzorul cu un potențiometrul: mai apare? De cele mai multe ori, defectul se lămurește singur în timpul acestei reduceri, pentru că partea eliminată la care defectul dispăre arată direct cauza.

Al doilea principiu este *măsoară, nu ghici*. Într-un sistem încorporat, majoritatea ipote-

zelor plauzibile sunt false, iar verificarea lor costă adesea mai puțin decât discuția despre ele. Dacă bănuiești că întreruperea nu se declanșează, comută un pin în rutina de întrerupere și privește-l cu osciloscopul; în trei minute ai un răspuns sigur. Dacă bănuiești că bucla principală durează prea mult, ridică un pin la intrarea în buclă și coboară-l la ieșire: lățimea impulsului este chiar durata buclei, iar spațiul dintre impulsuri este timpul liber.

Exemplu 10.6 Pin de test pentru măsurarea duratei buclei cu osciloscopul

```

1 #define PIN_TEST_SUS    (PORTD |=  (1 << PD7) )
2 #define PIN_TEST_JOS    (PORTD &= ~(1 << PD7) )
3
4 for (;;) {
5     if (bTic) {
6         bTic = 0;
7         PIN_TEST_SUS;                /* front crescator: intrarea in
           ↪ prelucrare */
8         eveniment_t ev = colecteazaEveniment();
9         ruleazaMasinaDeStari(ev);
10        PIN_TEST_JOS;                /* front cazator: prelucrarea s-a
           ↪ incheiat */
11    }
12 }
```

Această tehnică, numită uneori depanare cu un fir, are un avantaj pe care depanatorul cu puncte de oprire nu îl are: nu oprește programul. Într-un sistem care comandă un motor sau o valvă, oprirea execuției într-un punct de oprire schimbă complet comportamentul fizic, uneori periculos, iar defectele legate de timp dispar tocmai pentru că timpul a fost modificat de instrumentul de măsură. Comutarea unui pin costă două instrucțiuni și nu perturbă practic nimic.

Al treilea principiu este să se izoleze întrebarea „este hardware sau este software?” cât mai devreme, pentru că răspunsul schimbă complet direcția căutării. Un LED care nu se aprinde poate însemna un bit scris greșit sau un rezistor lipsă; măsurarea tensiunii pe pin separă imediat cele două situații. Dacă pinul are nivelul corect și LED-ul nu luminează, problema este în circuit; dacă pinul rămâne la zero, problema este în cod sau în configurarea direcției portului. Aceeași metodă se aplică la orice interfață.

Observație

Când un defect dispare după ce ai adăugat o instrucțiune de afișare pe interfața serială sau o întârziere, nu ai reparat nimic: ai modificat timpii. Asemenea defecte, sensibile la temporizare, provin aproape întotdeauna dintr-o variabilă partajată cu o rutină de întrerupere, fără calificatorul `volatile`, sau dintr-un acces neprotejat la o variabilă pe mai mulți octeți. Reparația corectă este protejarea accesului, nu păstrarea întârzierii care „face să meargă”.

10.11 Documentarea

O parte însemnată a documentației există deja, dacă etapele anterioare au fost parcurse: descrierea sistemului, caietul de sarcini, diagramele bloc, diagramele de activități și tabelele de tranziții, planul de testare și rezultatele încercărilor. Documentarea finală înseamnă strângerea lor într-un ansamblu coerent și completarea cu ceea ce lipsește pentru ca un alt inginer, care nu a participat la proiect, să îl poată prelua.

Acesta este criteriul practic al unei documentații suficiente: nu numărul de pagini, ci întrebarea dacă un coleg poate, folosind numai documentele, să repare un exemplar defect, să compileze programul, să îl încarce și să verifice că funcționează. Pentru a răspunde afirmativ, documentația trebuie să conțină cel puțin schema electrică a plăcii, cu valorile componentelor și cu numerele de catalog; harta pinilor, adică tabelul care spune ce este conectat la fiecare pin al microcontrolerului, cu direcția și cu nivelul activ; descrierea protocolului de comunicație, cu viteza, formatul cadrului și structura mesajelor; valorile de configurare, care includ atât parametrii aplicației, cât și biții de configurare ai microcontrolerului; și procedura de programare, adică ce program se folosește, ce interfață, ce comandă și cum se verifică rezultatul.

Tabela 10.5 Harta pinilor pentru sistemul de udare automată, fragment din documentația de proiect.

Pin	Direcție	Semnal	Observații
PC0 (ADC0)	Intrare	Senzor umiditate	0...5 V, filtru RC 10 k Ω / 100 nF
PB0	Intrare	Buton oprire	Activ în zero, rezistor intern activat
PB1	Intrare	Buton udare manuală	Activ în zero, rezistor intern activat
PB2	Intrare	Buton resetare avarie	Activ în zero, rezistor intern activat
PD5	Ieșire	Comandă releu valvă	Activ în unu, prin tranzistor și diodă de descărcare
PD6	Ieșire	LED udare	Activ în unu, rezistor 330 Ω
PD4	Ieșire	LED avarie	Activ în unu, rezistor 330 Ω
PD7	Ieșire	Pin de test	Numai pentru depanare, neconectat în producție
PD0, PD1	Serial	Jurnal RXD, TXD	9600 bd, 8N1

Un capitol aparte îl constituie documentarea codului. Fiecare funcție trebuie însoțită de o descriere a scopului ei, a parametrilor de intrare cu unitățile și domeniile lor, a valorii returnate și a condițiilor de eroare, iar dacă înăuntru se aplică un algoritm netrivial, acesta trebuie explicat în cuvinte, pentru că din cod nu se poate deduce intenția. Comentariile care repetă instrucțiunea nu au valoare; cele care explică de ce s-a ales o anumită soluție sunt

cele care salvează timp peste doi ani.

Exemplu 10.7 Antet de funcție documentat: intrari, iesiri, erori, algoritmi

```

1  /**
2   * Converteste rezultatul brut al conversiei ADC in umiditate
      ↳ relativa .
3   *
4   * Algoritm: interpolare liniara intre doua puncte de calibrare
      ↳ masurate
5   * la punerea in functiune , sol uscat si sol saturat .
      ↳ Caracteristica
6   * senzorului este descrescatoare , deci panta este negativa .
7   *
8   * @param brut    rezultatul conversiei , 0..1023
9   * @param uscat   valoarea bruta masurata in sol uscat (punct de
      ↳ calibrare)
10  * @param umed    valoarea bruta masurata in sol saturat (punct de
      ↳ calibrare)
11  * @return        umiditate in procente x 10, domeniul 0..1000;
12  *               valoarea 0xFFFF daca argumentul este in afara
      ↳ domeniului
13  *               util , adica senzor deconectat sau in scurtcircuit
14  */
15 uint16_t adcLaUmiditate(uint16_t brut , uint16_t uscat , uint16_t
      ↳ umed) ;

```

Numele variabilelor și ale funcțiilor fac parte din documentație. Un nume sugestiv economisește un comentariu: `durataMaximaUdareSecunde` spune tot, în timp ce `d` nu spune nimic și obligă cititorul să caute declarația. Multe colective folosesc notații consacrate, în care numele poartă un prefix ce indică tipul sau natura variabilei, de exemplu `nContor` pentru un întreg, `bActiv` pentru un indicator logic sau `pTampon` pentru un pointer. Convenția în sine contează mai puțin decât consecvența cu care este aplicată în tot proiectul.

Documentației trebuie să i se adauge, în final, ceea ce se numește uneori jurnalul deciziilor: lista scurtă a alegerilor importante și a motivelor lor. De ce s-a ales histerezisul de 5 la sută și nu de 2. De ce pauza obligatorie este de 30 de minute. De ce releul este comandat prin tranzistor și nu direct. Fiecare dintre aceste decizii pare evidentă în ziua în care este luată și devine de neînțeles peste un an, inclusiv pentru cel care a luat-o. Fără jurnal, primul lucru pe care îl face cel care preia proiectul este să „corecteze” o decizie corectă, iar defectul reapare exact în situația care a impus-o inițial.

Exemplu

Se cere estimarea rezervei de timp de procesor a sistemului de udare, pentru a verifica dacă bucla ritmată la 10 ms este realizabilă. Măsurarea cu pinul de test arată un impuls de $480 \mu\text{s}$ la ticurile obișnuite și de 1,3 ms la ticul în care se face conversia ADC cu mediere.

Rezolvare. Perioada ticului este de 10 ms, deci timpul disponibil într-un ciclu este de 10 ms. În cazul cel mai defavorabil, prelucrarea consumă 1,3 ms, adică $1,3/10 = 0,13$, deci 13 la sută din timpul disponibil. Rezerva este de 87 la sută, cu mult peste minimul recomandat de 20 la sută. Sistemul are loc pentru funcții suplimentare, iar dacă cerințele nu cresc, frecvența ceasului poate fi chiar redusă pentru a scădea consumul, ceea ce ajută la respectarea cerinței CN1. Observăm că această concluzie a rezultat dintr-o măsurătoare de trei minute, nu dintr-o estimare, și că ea a fost posibilă numai pentru că pinul de test fusese prevăzut de la început în harta pinilor.

Întrebări de verificare

1. Explicați de ce o cerință de forma „sistemul trebuie să răspundă repede la apăsarea butonului” nu poate fi acceptată într-un caiet de sarcini și rescrieți-o astfel încât să devină verificabilă, precizând și metoda de măsurare.
2. Dați trei exemple de cerințe nefuncționale pentru un sistem de irigare alimentat de la un panou fotovoltaic și arătați, pentru fiecare, cum influențează alegerea microcontrollerului și a schemei electrice.
3. De ce interfața unui subsistem trebuie fixată înainte de implementarea lui? Descrieți ce informații trebuie să conțină descrierea completă a unei interfețe, dincolo de prototipul funcției.
4. Un coleg propune ca funcția care decide pornirea udării să citească singură convertorul analog-numeric și să comande direct releul, „ca să fie mai puține funcții”. Explicați ce se pierde din punctul de vedere al testării și cum rescrieți codul.
5. Construiți un tabel de criterii pentru alegerea microcontrollerului unui sistem care măsoară patru temperaturi, comandă două motoare prin PWM și comunică prin interfața I2C cu un afișaj. Indicați numărul minim de pini și perifericele necesare.
6. Comparați bucla de scrutare cu arhitectura condusă de evenimente din punctul de vedere al timpului maxim de reacție și al complexității codului. În ce situație un sistem cu buclă simplă nu poate respecta o cerință de întârziere maximă?
7. Completați tabelul de tranziții din acest capitol cu o stare nouă, CALIBRARE, în care sistemul memorează valorile pentru sol uscat și sol saturat. Precizați toate tranzițiile de intrare și de ieșire și acțiunile asociate.

8. Explicați rolul histerezisului în funcția de decizie a udării. Ce se întâmplă fizic dacă histerezisul este zero, iar semnalul de la senzor are un zgomot de câteva unități în jurul pragului?
9. Un sistem folosește un contor de milisecunde pe 16 biți pentru a măsura o pauză de 30 de minute. Arătați de ce implementarea este greșită, calculați după cât timp apare eroarea și propuneți două soluții.
10. Descrieți procedura prin care ați determina, cu un osciloscop și un singur pin liber, durata maximă a buclei principale și frecvența reală a ticului de sistem. Ce concluzii se pot trage din raportul dintre cele două valori?
11. Un defect apare o dată la câteva minute și dispare dacă adăugați o instrucțiune de afișare pe interfața serială. Explicați ce clasă de erori este cel mai probabil implicată, de ce afișarea o maschează și cum se repară corect.
12. Enumerați documentele minime pe care trebuie să le primească un inginer care preia un proiect terminat și arătați, pentru fiecare, ce anume nu ar putea face fără el.

11. Studiu de caz: robotul GT

Contextul și tema de proiect
Caietul de sarcini
Partiționarea sistemului
Bugetul de resurse
Arhitectura programului
Comanda motoarelor
Odometria din encodere
Urmărirea liniei
Planul de testare
Ce a mers și ce nu
Întrebări de verificare

11.1 Contextul și tema de proiect

Capitolul precedent a descris o metodă: descrierea sistemului, caietul de sarcini, partiționarea în subsisteme, detalierea lor, scrierea codului, testarea pe niveluri și documentarea. Metoda a fost ilustrată acolo pe un exemplu mic, cu un singur senzor și o singură ieșire de putere. Un exemplu mic are avantajul clarității, dar ascunde tocmai ceea ce face dificilă proiectarea reală: concurența dintre subsisteme pentru aceleași resurse ale microcontrolerului, timpii care nu mai încap unul lângă altul și compromisurile care apar atunci când cerințele sunt mai multe decât pinii.

Capitolul de față reia aceeași metodă pe un proiect care are aceste dificultăți: un *robot mobil autonom*, construit pentru un concurs studentesc. Contextul competițional impune, din afară, exact ceea ce lipsește de obicei din proiectele didactice: un caiet de sarcini scris de altcineva, pe care nu îl poți negocia, un termen fix și o probă publică în care sistemul fie funcționează, fie nu. Organizatorii publică regulamentul, dimensiunile terenului, caracteristicile marcajelor, limitele de gabarit și de masă, precum și componentele obligatorii de siguranță: butonul de oprire de urgență accesibil din exterior, sistemul de oprire automată la pierderea legăturii sau la ieșirea din zona de joc și alimentarea de la acumulatori proprii. Toate acestea intră direct în caietul de sarcini, fără discuție.

Robotul pe care îl vom urmări are o construcție simplă și tipică: șasiu pe patru roți, dintre care două motoare, cu tracțiune diferențială, adică viraj obținut prin diferența de

viteză dintre roata stângă și cea dreaptă. El trebuie să parcurgă autonom un traseu marcat cu bandă pe podea, să ocolească obstacolele apărute, să raporteze permanent starea proprie unei console de depanare și să poată fi oprit oricând de la o telecomandă. Este, cu alte cuvinte, un sistem în care intervin aproape toate modulele studiate până acum: porturile pentru comenzile logice și pentru butoane, întreruperile pentru evenimentele rapide, modulele de tip timer pentru comanda în PWM și pentru baza de timp, convertorul analog-numeric pentru senzorii de linie și pentru supravegherea acumulatorului, interfața serială asincronă pentru comunicație.

Ceea ce face greu un asemenea proiect nu este partea de algoritmică, modestă aici la un regulator proporțional și o mașină cu cinci stări, ci încadrarea în resurse și respectarea timpilor. Un ATmega328P oferă douăzeci de pini utilizabili pe o placă Arduino Uno, trei module de tip timer, un singur convertor analog-numeric multiplexat, un singur USART și două intrări de întrerupere externă adevărate. Vom vedea că cererea depășește oferta aproape la fiecare capitol și că proiectarea constă, în bună măsură, în alegerea compromisurilor.

11.2 Caietul de sarcini

Regulamentul concursului este descrierea sistemului în sensul capitolului precedent: un document scris în limbaj obișnuit, de către cineva care nu proiectează electronică. El spune, de exemplu, că robotul „trebuie să urmărească traseul marcat” și că „trebuie să evite coliziunile”. Aceste formulări nu sunt verificabile. Primul act tehnic al echipei este transformarea lor într-un caiet de sarcini cu cerințe măsurabile, adică afirmații pe care două persoane independente, cu un cronometru și o riglă, le pot declara îndeplinite sau nu.

Trecerea aceasta cere întotdeauna întrebări către organizator, iar întrebările sunt cele care fixează cifrele. Cât de lată este banda de marcaj? Care este raza minimă de curbura a traseului? Traseul are întreruperi sau intersecții? Obstacolele sunt fixe sau pot apărea în mișcare? Care este iluminarea din sală și poate ea varia în timpul probei? Fiecare răspuns devine o cerință, iar fiecare întrebare nepusă devine, mai târziu, o surpriză pe teren. Tabelul 11.1 prezintă un fragment din caietul de sarcini rezultat, cu cerințele funcționale notate CF și cele nefuncționale notate CN.

Se observă că fiecare cerință are un cod, o valoare numerică și o metodă de verificare. Cerințele CF4, CN3 și CN4 nu apar nicăieri în regulamentul concursului: ele au rezultat din lista de întrebări despre situațiile anormale. Cerința CN4 în special merită atenție, pentru că exprimă un principiu de siguranță care nu este evident: un senzor deconectat produce, la intrarea convertorului analog-numeric, o valoare nedefinită, iar dacă algoritmul o folosește ca și cum ar fi validă, rezultatul poate fi o comandă maximă pe un motor, adică exact comportamentul periculos. Cerința obligă la o verificare de plauzibilitate a fiecărei mărimi măsurate înainte de a intra în calcul.

Tabela 11.1 Fragment din caietul de sarcini al robotului, cu criterii de acceptare măsurabile.

Cod	Enunțul cerinței	Criteriu de acceptare
CF1	Robotul urmărește o bandă neagră de 19 mm lățime pe fond deschis, cu raza minimă de curbură de 250 mm.	Zece tururi complete fără pierderea traseului.
CF2	Viteza medie pe un tur complet este de cel puțin 0,4 m/s.	Cronometrarea a cinci tururi consecutive.
CF3	La apariția unui obstacol la mai puțin de 250 mm în față, robotul se oprește în cel mult 300 ms și trece în procedura de ocolire.	Măsurarea distanței de oprire, 20 de repetări.
CF4	Dacă banda nu mai este detectată timp de 2 s, robotul oprește motoarele și semnalizează.	Ridicarea robotului de pe pistă, cronometrare.
CF5	La acționarea butonului de urgență sau la primirea comenzii de oprire, ambele motoare sunt aduse la zero în cel mult 50 ms, din orice stare.	Osciloscop pe buton și pe comanda punții, 20 de repetări.
CF6	Robotul transmite pe interfața serială, la fiecare 100 ms, starea curentă, eroarea de poziție, distanțele măsurate și vitezele roților.	Înregistrarea jurnalului timp de 5 minute.
CF7	Odometria estimează distanța parcursă pe un traseu drept de 2 m cu o eroare sub 3 la sută.	Zece încercări, comparație cu ruleta.
CN1	Ciclul de reglare are perioada de 10 ± 1 ms, cu o încărcare a procesorului sub 50 la sută.	Măsurare cu osciloscopul pe pinul de test.
CN2	Autonomia este de minimum 20 de minute de funcționare continuă.	Încercare cu acumulatorul încărcat complet.
CN3	Sub tensiunea minimă a acumulatorului, robotul se oprește controlat și semnalizează.	Alimentare de la sursă reglabilă, coborâre lentă.
CN4	Niciun defect al unui senzor nu produce o comandă de motor mai mare decât cea din regimul normal.	Deconectarea pe rând a fiecărui senzor.

Observație

Componentele obligatorii impuse de regulament, butonul de urgență și oprirea automată, nu trebuie să depindă numai de software. Butonul de urgență corect proiectat întrerupe fizic alimentarea etajului de putere, printr-un contact în serie cu puntea H, și abia în al doilea rând anunță microcontrolerul printr-o intrare digitală. Un buton care doar aprinde un bit citit de program nu protejează în cazul care contează cel mai mult, acela în care programul s-a blocat.

11.3 Partiționarea sistemului

Cu cerințele fixate, se trece la împărțirea ansamblului în subsisteme cu responsabilitate unică. Regula din capitolul precedent rămâne valabilă: interfața fiecărui subsistem se scrie înaintea implementării lui, astfel încât toate subsistemele să poată fi dezvoltate în paralel de membrii echipei. La un robot, partiționarea urmează aproape de la sine funcțiile fizice, iar rezultatul este arătat în Figura 11.1.

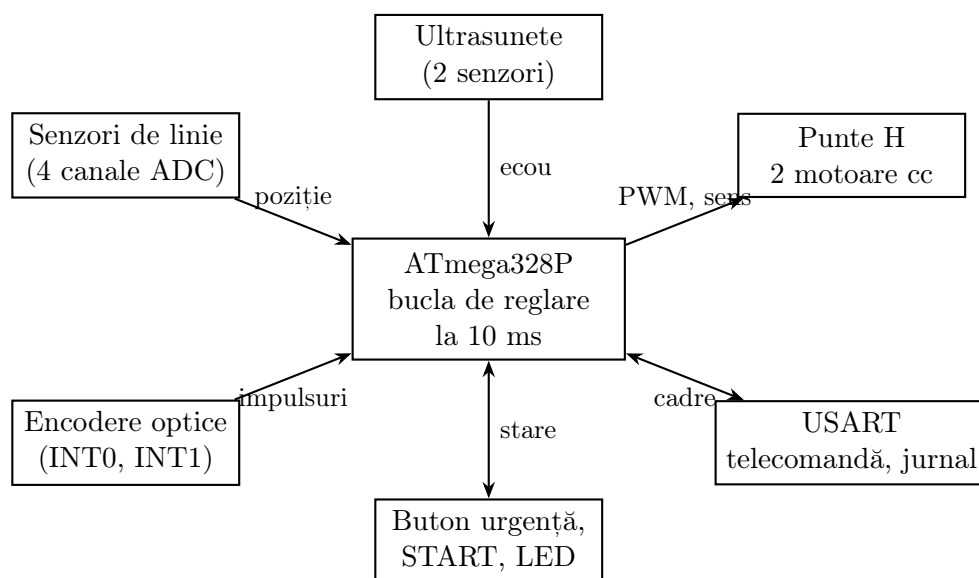


Figura 11.1 Schema bloc a robotului. Microcontrolerul stă în centru, iar fiecare subsistem comunică cu el printr-o interfață cu mărimi bine definite.

Această figură este mai mult decât un desen: ea este harta responsabilităților. Subsistemul senzorilor de linie livrează o singură mărime, eroarea de poziție, și ascunde complet faptul că în spate sunt patru fototranzistoare și patru conversii analog-numerice. Subsistemul de odometrie livrează două viteze, în milimetri pe secundă, și ascunde faptul că ele provin din impulsuri numărate în întreruperi. Subsistemul de tracțiune primește două comenzi cu semn și ascunde puntea H, sensul de rotație și zona moartă a motoarelor. Dacă mâine se schimbă tipul senzorului de linie sau modelul punții, se rescrie un singur subsistem, iar restul programului nu află nimic.

Tabelul 11.2 leagă fiecare subsistem de modulul de microcontroler pe care se sprijină și de capitolul în care acel modul a fost studiat. Este util ca un asemenea tabel să existe în

documentația oricărui proiect: el arată dintr-o privire ce cunoștințe sunt necesare pentru a prelua proiectul și, mai ales, ce se strică dacă un modul este reconfigurat.

Tabela 11.2 Legătura dintre subsistemele robotului, modulele microcontrolerului și capitolele în care acestea au fost tratate.

Subsistem	Ce face	Modul folosit	Capitolul
Tracțiune	Comandă turația și sensul celor două motoare	Timer1 în PWM rapid, porturi digitale	6 și 4
Odometrie	Numără impulsurile encoderelor și calculează vitezele	Întreruperi externe INT0, INT1	5
Percepția distanței	Măsoară lățimea impulsului de ecou	Întreruperi la schimbarea pinului, baza de timp	5 și 6
Percepția liniei	Citește patru senzori reflexivi	Convertorul analog-numeric	7
Supravegherea alimentării	Măsoară tensiunea acumulatorului	Convertorul analog-numeric	7
Comunicație	Primește comenzi, trimite jurnalul	USART asincron	8
Baza de timp	Produce ticul de 10 ms al buclei	Timer2 în mod CTC	6
Interfața cu operatorul	Buton de urgență, buton de start, LED de stare	Porturi de intrare-ieșire	4

Ultima coloană conține o absență semnificativă: nu apare niciun subsistem care să folosească magistralele sincrone studiate în capitolul 9. Echipa a dorit inițial un afișaj și un senzor inerțial pe interfața I²C, dar ambele au fost eliminate. Motivul îl vom vedea imediat: nu au mai rămas pini.

11.4 Bugetul de resurse

Alegerea microcontrolerului s-a făcut, în cazul acesta, sub o constrângere suplimentară: placa Arduino Uno era deja disponibilă, iar echipa o cunoștea. Aceasta este o situație frecventă în practică și nu este în sine greșită, cu condiția de a verifica explicit că piesa aleasă acoperă necesarul. Verificarea se face întocmind un *buget de resurse*, adică o listă a tuturor resurselor cerute de subsisteme, comparată cu ceea ce oferă piesa. Bugetul se face pe patru capitole: pini, module de tip timer, canale analogice și surse de întrerupere.

11.4.1 Pini

Placa Arduino Uno pune la dispoziție douăzeci de pini de intrare-ieșire, restul liniilor microcontrolerului fiind ocupate de cuarț și de circuitul de resetare. Tabelul 11.3 arată repartizarea lor finală. Se observă un lucru neplăcut: nu a rămas niciunul liber.

Tabela 11.3 Harta pinilor robotului. Toate cele douăzeci de linii disponibile sunt ocupate.

Pin	Direcție	Semnal
PD0, PD1	USART	Recepția comenzilor și transmisia jurnalului
PD2	Intrare	INT0, canalul A al encoderului stâng
PD3	Intrare	INT1, canalul A al encoderului drept
PD4	Ieșire	Sensul motorului stâng
PD5	Ieșire	Declanșarea senzorului cu ultrasunete stâng
PD6	Ieșire	Declanșarea senzorului cu ultrasunete drept
PD7	Ieșire	Sensul motorului drept
PB0	Intrare	PCINT0, ecoul senzorului stâng
PB1	Ieșire	OC1A, PWM pentru motorul stâng
PB2	Ieșire	OC1B, PWM pentru motorul drept
PB3	Intrare	PCINT3, ecoul senzorului drept
PB4	Intrare	Butonul de start, activ în zero
PB5	Ieșire	LED de stare, folosit și ca pin de test
PC0...PC3	Analogică	Cei patru senzori de linie
PC4	Analogică	Tensiunea acumulatorului, prin divizor rezistiv
PC5	Intrare	Contactul butonului de urgență, activ în zero

Consecințele acestei saturații sunt trei, și toate au fost decizii conștiente. Prima: senzorii de linie sunt patru, nu șase cum ar fi fost de dorit, ceea ce înseamnă că poziția liniei se cunoaște mai grosier, iar la viteze mari robotul taie curbele. A doua: interfața I²C dispare, pentru că liniile ei, PC4 și PC5, au fost date senzorului de tensiune și butonului de urgență; odată cu ea dispar afișajul și senzorul inerțial. A treia, cea mai supărătoare: nu mai există un pin de test liber, deși capitolul precedent insista că un asemenea pin este instrumentul cel mai ieftin de depanare. Soluția de compromis a fost folosirea pinului LED-ului de stare și în acest scop, cu prețul unui LED care clipește dezordonat în timpul măsurătorilor.

Un al doilea conflict, mai subtil, apare la comunicație. Cerințele cer două lucruri diferite pe interfața serială: o consolă de depanare, care primește jurnalul, și o telecomandă, care trimite comenzi. ATmega328P are însă un singur USART. Există trei ieșiri din această situație. Se poate implementa un al doilea USART în software, pe o pereche de pini oarecare, dar recepția software cere dezactivarea întreruperilor pe durata fiecărui caracter, ceea ce introduce întârzieri în numărarea impulsurilor de la encodere, adică strică odometria. Se

poate renunța la una dintre funcții. Sau, soluția aleasă aici, cele două funcții pot împărți aceleași linii: receptorul telecomenzii injectează cadrele pe linia RXD, iar jurnalul iese pe TXD, unde este ascultat de consola de depanare, conectată numai în laborator. Nu se pierde nimic, pentru că în timpul probei nimeni nu citește oricum consola.

11.4.2 Timerele, canalele analogice și întreruperile

Cele trei module de tip timer sunt de asemenea complet ocupate. *Timer1* generează cele două semnale PWM pentru motoare, în modul PWM rapid pe opt biți, cu prescalarea 8. Frecvența rezultată este $16\,000\,000/(8 \cdot 256) = 7812,5$ Hz, adică peste domeniul audibil, ceea ce elimină fluieratul supărător al motoarelor. *Timer2* produce ticul de 10 ms al buclei de reglare, în mod CTC, cu prescalarea 1024 și cu registrul de comparare încărcat cu 155: perioada este $1024 \cdot 156/16\,000\,000 = 9,98$ ms, suficient de aproape de 10 ms. *Timer0* rămâne bazei de timp în microsecunde, folosită la măsurarea ecourilor.

Această ultimă alegere merită explicată, pentru că ea contrazice ceea ce ar sugera manualul. Măsurarea corectă a lățimii unui impuls se face cu modulul de captură al lui *Timer1*, pe pinul ICP1, care memorează automat valoarea numărătorului la apariția frontului, fără nicio incertitudine legată de latența întreruperii. Aici însă *Timer1* este configurat pentru PWM pe opt biți, deci numărătorul lui se rotește la fiecare $128\ \mu\text{s}$, în timp ce un ecou poate dura până la circa 17,5 ms. Captura ar deveni inutilizabilă fără numărarea separată a depășirilor. Compromisul acceptat a fost măsurarea ecoului prin citirea bazei de timp în microsecunde la cele două fronturi, cu o rezoluție de $4\ \mu\text{s}$. Cum sunetul parcurge dus-întors circa 1 cm în $58\ \mu\text{s}$, cei $4\ \mu\text{s}$ înseamnă sub un milimetru, adică mult mai puțin decât eroarea proprie a sensorului. Compromisul este deci acceptabil, iar concluzia aceasta se sprijină pe un calcul, nu pe o impresie.

Convertorul analog-numeric are șase canale accesibile, dintre care patru sunt folosite pentru senzorii de linie și unul pentru tensiunea acumulatorului. La prescalarea 128, ceasul convertorului este de 125 kHz, iar o conversie durează 13 perioade, adică $104\ \mu\text{s}$. Cele cinci conversii dintr-un ciclu consumă circa $520\ \mu\text{s}$, adică peste cinci la sută din perioada de 10 ms a buclei. Nu este mult, dar nici neglijabil, iar dacă am fi vrut cei șase senzori de linie doriți inițial, costul ar fi crescut proporțional.

Sursele de întrerupere folosite sunt cinci: INT0 și INT1 pentru cele două encodere, vectorul de schimbare de pin al portului B pentru cele două ecouri, comparația lui *Timer2* pentru ticul de sistem și recepția USART pentru comenzi. Trebuie reținut, din capitolul dedicat întreruperilor, că întreruperile la schimbarea pinului au un singur vector pentru un întreg port: rutina nu știe care pin s-a schimbat și trebuie să deducă acest lucru comparând valoarea curentă a registrului de intrare cu cea memorată anterior. În plus, ea se declanșează pe ambele fronturi, ceea ce în cazul de față este chiar util, pentru că avem nevoie de amândouă.

Exemplu

Se cere estimarea încărcării procesorului produse numai de numărarea impulsurilor de la encodere, la viteza maximă de deplasare. Roțile au diametrul de 80 mm, discurile encoderelor au 60 de fante, iar viteza maximă proiectată este de 0,6 m/s. Rutina de întrerupere durează, măsurat, circa $4 \mu\text{s}$, inclusiv salvarea și refacerea contextului.

Rezolvare. Circumferința roții este $\pi \cdot 80 = 251,3$ mm, iar cu 60 de fante rezultă un impuls la fiecare $251,3/60 = 4,19$ mm de drum parcurs. La 0,6 m/s, adică 600 mm/s, o roată produce $600/4,19 = 143$ impulsuri pe secundă. Numărând pe un singur front și având două roți, se ajung la 286 de întreruperi pe secundă, deci $286 \cdot 4 \mu\text{s} = 1,14$ ms de procesor consumate în fiecare secundă, adică 0,11 la sută. Chiar dacă am număra ambele fronturi ale semnalului, pentru a dubla rezoluția la 2,09 mm, încărcarea ar rămâne sub un sfert de procent. Concluzia este că numărarea în întrerupere este practic gratuită, ceea ce justifică alegerea ei în locul scrutării.

11.5 Arhitectura programului

Programul robotului este organizat după schema hibridă descrisă în capitolul precedent: întreruperile captează evenimentele rapide și marchează timpul, iar o buclă principală ritmată de un tic ia toate deciziile. Perioada ticului, de 10 ms, nu a fost aleasă la întâmplare. Ea trebuie să fie suficient de scurtă pentru ca reglajul să fie mai rapid decât dinamica robotului, dar suficient de lungă pentru ca toate operațiile unui ciclu să încapă cu rezervă. Constanta de timp mecanică a unui motor mic de curent continuu cu reductor este de ordinul zecilor de milisecunde, deci un ciclu de 10 ms dă zeci de corecții pe durata unei manevre, ceea ce este comod.

Regula de aur a acestei bucle este că nimic lent nu are voie să intre în ea. Motivul este cantitativ și se vede imediat pe cerințe. Dacă bucla ar aștepta, într-un ciclu, ecoul unui senzor cu ultrasunete, ar sta blocată până la 17,5 ms, adică mai mult decât întreaga perioadă a ticului; reglajul liniei s-ar întrerupe, robotul ar merge orb și ar ieși de pe traseu în curbă. Dacă bucla ar transmite sincron cadrul de jurnal, cu așteptarea golirii registrului, ar pierde, la 38 400 bd, circa 0,26 ms pentru fiecare caracter, deci peste 10 ms pentru un cadru de patruzeci de caractere. De aceea măsurarea ecoului se face prin întreruperi, iar jurnalul se depune într-un tampon circular din care rutina de întrerupere a transmițătorului scoate câte un octet.

Testul practic prin care se verifică dacă regula a fost respectată este măsurarea, cu osciloscopul, a lățimii impulsului produs pe pinul de test la intrarea și la ieșirea din prelucrare. La robotul discutat aici, lățimea măsurată a fost de circa 1,1 ms, adică unsprezece la sută din perioada de 10 ms, cu mult sub limita de cincizeci la sută cerută de CN1. Rezerva rămasă este ceea ce a permis, mai târziu, adăugarea odometriei fără reproiectarea buclei.

11.5.1 Structura pe mașină de stări

Comportamentul robotului nu poate fi descris printr-o singură lege de reglare, pentru că aceeași citire de senzor înseamnă lucruri diferite în situații diferite. O sumă mică pe senzorii de linie înseamnă „linia s-a pierdut” dacă robotul o urmărea, dar este perfect normală dacă robotul ocolește un obstacol și se află momentan lângă traseu. Forma corectă de organizare este, ca și în capitolul precedent, *mașina cu stări finite*, arătată în Figura 11.2.

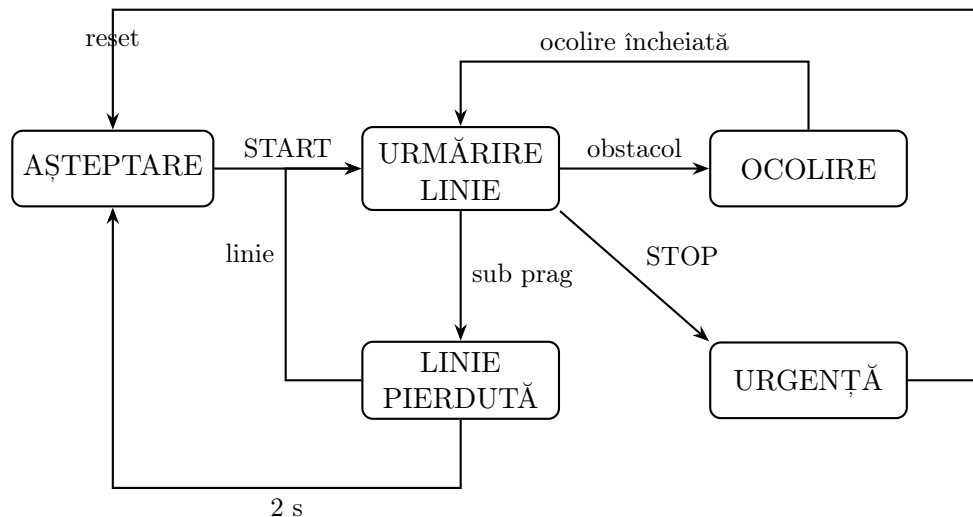


Figura 11.2 Mașina cu stări a robotului. Tranziția către starea de URGENTĂ, desenată o singură dată pentru claritate, este posibilă din orice stare.

Câteva decizii de proiectare sunt ascunse în această diagramă și merită scoase la lumină. Nu există tranziție directă din OCOLIRE în LINIE PIERDUTĂ: pe durata ocolirii, absența liniei este starea normală și nu trebuie să declanșeze oprirea. Nu există tranziție directă din LINIE PIERDUTĂ în OCOLIRE: dacă robotul nu știe unde este traseul, nu are sens să înceapă o manevră relativă la el. Ieșirea din URGENTĂ nu se face automat la eliberarea butonului, ci cere o acțiune explicită a operatorului, pentru ca un contact imperfect să nu re-poarnească motoarele. Fiecare dintre aceste afirmații este o cerință de siguranță transformată în absența unei tranziții, formă mult mai ușor de verificat decât o condiție complicată.

Codul care rezultă este o transcriere directă a diagramei. Se observă în listingul următor că verificarea condițiilor de urgență se face înaintea instrucțiunii de selecție, deci se aplică din orice stare, și că ramura implicită duce sistemul în starea sigură: dacă variabila de stare ajunge, dintr-un motiv oarecare, la o valoare imposibilă, robotul se oprește.

Exemplu 11.1 Bucla principală ritmata la 10 ms, organizată ca mașina cu stări

```

1 typedef enum {
2     ST_AȘTEPTARE = 0,
3     ST_URMARIRE,
4     ST_OCOLIRE,
5     ST_LINIE_PIERDUTA,
6     ST_URGENTA

```

```
7 } stare_t;
8
9 static stare_t stare = ST_ASTEPTARE;
10 static cronometru_t tCautare;          /* temporizarea cautarii
    ↪ liniei */
11 volatile uint8_t bTic = 0;            /* ridicat de ISR-ul lui
    ↪ Timer2 */
12
13 int main(void)
14 {
15     motoareOprireImediata();          /* stare sigura INAINTE de
    ↪ configurare */
16     porturiInit();
17     pwmInit();                        /* Timer1, PWM rapid pe 8 biti
    ↪ */
18     ticInit();                        /* Timer2, CTC, intrerupere la
    ↪ 10 ms */
19     encodereInit();                  /* INT0 si INT1, front
    ↪ crescator */
20     ultrasuneteInit();                /* PCINT0 pe PB0 si PB3 */
21     adcInit();
22     usartInit(38400);
23     sei();
24
25     for (;;) {
26         if (!bTic) {
27             continue;                /* asteptam ticul; nicio
    ↪ intarziere activa */
28         }
29         bTic = 0;                     /* scriere pe un octet:
    ↪ atomica pe AVR */
30
31         odometrieActualizeaza();      /* citeste contoarele,
    ↪ calculeaza vitezele */
32         senzoriLinieCiteste();         /* 4 conversii ADC, circa 420
    ↪ us */
33         ultrasunetePlanifica();       /* declanseaza alternativ cate
    ↪ un senzor */
34         alimentareVerifica();
35
36         if (butonUrgenta() || comandaStop()) {
```

```
37         motoareOprireImediata();
38         stare = ST_URGENTA;
39     }
40
41     switch (stare) {
42
43     case ST_ASTEPTARE:
44         motoareComanda(0, 0);
45         if (butonStart() || comandaStart()) {
46             stare = ST_URMARIRE;
47         }
48         break;
49
50     case ST_URMARIRE:
51         if (distantaFata() < D_OBSTACOL_MM) {
52             ocolireIncepe();
53             stare = ST_OCOLIRE;
54         } else if (!linieVizibila()) {
55             cronometruPornire(&tCautare, 2000);
56             stare = ST_LINIE_PIERDUTA;
57         } else {
58             regulatorLinie();
59         }
60         break;
61
62     case ST_OCOLIRE:
63         ocolirePas();
64         if (ocolireTerminata() && linieVizibila()) {
65             stare = ST_URMARIRE;
66         }
67         break;
68
69     case ST_LINIE_PIERDUTA:
70         cautaLinia(); /* rotire lenta in ultimul
71                      ↪ sens cunoscut */
72         if (linieVizibila()) {
73             stare = ST_URMARIRE;
74         } else if (cronometruExpirat(&tCautare)) {
75             motoareComanda(0, 0);
76             stare = ST_ASTEPTARE;
77         }
```

```

77         break;
78
79     case ST_URGENTA:
80         motoareOprireImediata();
81         if (butonReset() && !butonUrgenta()) {
82             stare = ST_ASTEPTARE;
83         }
84         break;
85
86     default:                                     /* stare imposibila: oprire
87         ↪ preventiva */
88         motoareOprireImediata();
89         stare = ST_URGENTA;
90         break;
91     }
92     jurnalUmpleTampon();                         /* depune octeti , nu asteapta
93         ↪ transmisia */
94 }

```

11.6 Comanda motoarelor

Motoarele de curent continuu nu se comandă în tensiune continuă variabilă, ci în *PWM*, adică printr-o tensiune comutată între zero și tensiunea acumulatorului, la frecvență fixă și cu factor de umplere variabil. Inductanța înfășurării netezește curentul, iar motorul se comportă ca și cum ar primi tensiunea medie. Avantajul, discutat în capitolul dedicat modulelor de tip timer, este că elementul de comutație lucrează numai în saturație sau blocat, deci disipă puțin, spre deosebire de un etaj liniar care ar transforma în căldură diferența de tensiune. Sensul de rotație se obține inversând polaritatea aplicată motorului, ceea ce cere o *punte H*, adică patru comutatoare care leagă bornele motorului fie într-un sens, fie în celălalt. Regula absolută la comanda unei punți este ca cele două brațe ale aceleiași ramuri să nu conducă niciodată simultan, pentru că aceasta ar însemna un scurtcircuit direct pe acumulator.

Trei fenomene fizice trebuie tratate în funcția de comandă, iar ignorarea lor produce exact defectele pe care echipele începătoare le observă pe teren fără să le înțeleagă.

Primul este *zona moartă*. Sub un anumit factor de umplere, cuplul produs de motor nu învinge frecarea statică din reductor și motorul rămâne nemișcat, deși consumă curent și se încălzește. Pragul se determină experimental, crescând comanda din unitate în unitate până când roata ridicată de pe sol începe să se rotească. Dacă nu este compensat, regulatorul de linie va produce, pentru erori mici, comenzi care nu se materializează în nicio mișcare, iar robotul va părea că ignoră abaterile mici și reacționează brusc la cele mari. Compensarea

constă în comprimarea domeniului util al comenzii peste intervalul dintre pragul măsurat și valoarea maximă, astfel încât orice comandă nenulă să producă mișcare.

Al doilea este *saltul de comandă*. O trecere bruscă de la zero la comanda maximă cere un curent limitat doar de rezistența înfășurării, deci mult mai mare decât cel nominal. Efectele sunt căderea tensiunii pe acumulator, care poate reseta microcontrolerul, patinarea roților, care strică odometria, și solicitarea mecanică a reductorului. Soluția este o *rampă*, adică limitarea variației comenzii la o valoare fixă pe ciclu. Cu un pas de opt unități pe ciclul de 10 ms, trecerea de la zero la comanda maximă de 255 durează $255/8 \cdot 10 = 319$ ms, ceea ce este imperceptibil în comportarea robotului, dar suficient pentru a elimina vârful de curent.

Al treilea este *oprirea de urgență*, care trebuie să ocolească deliberat rampa. Cerința CF5 cere aducerea motoarelor la zero în cel mult 50 ms, iar rampa ar avea nevoie de peste 300 ms. De aceea funcția de oprire imediată scrie direct zero în registrele de comparare, fără să treacă prin limitarea de variație. Aceasta este o excepție justificată de siguranță, iar faptul că este o excepție trebuie scris în comentariu, altfel un cititor ulterior o va considera o inconsecvență și o va „corecta”.

Exemplu 11.2 Comanda celor doua motoare: sens, zona moarta si rampa de acceleratie

```

1  #define PWM_MIN      40      /* masurat: sub acest prag motorul nu
    ↪ se roteste */
2  #define PWM_MAX      255
3  #define RAMPA_PAS     8      /* variatie maxima a comenzii intr-un
    ↪ tic de 10 ms */
4
5  static int16_t cmdStangaAplicata = 0;
6  static int16_t cmdDreaptaAplicata = 0;
7
8  /* Comprima domeniul util 1..255 peste intervalul PWM_MIN..PWM_MAX
    ↪ , astfel
9     incat orice comanda nenula sa produca miscare reala. */
10 static uint8_t pwmDinComanda(int16_t modul)
11 {
12     if (modul <= 0) {
13         return 0;
14     }
15     if (modul > 255) {
16         modul = 255;
17     }
18     return (uint8_t)(PWM_MIN + ((uint32_t)modul * (PWM_MAX -
    ↪ PWM_MIN)) / 255);
19 }
20
```

```

21 static int16_t aplicaRampa(int16_t curent, int16_t dorit)
22 {
23     if (dorit > curent + RAMPA_PAS) return curent + RAMPA_PAS;
24     if (dorit < curent - RAMPA_PAS) return curent - RAMPA_PAS;
25     return dorit;
26 }
27
28 /* cmdStanga, cmdDreapta: -255..+255; semnul da sensul de rotatie.
   ↪ */
29 void motoareComanda(int16_t cmdStanga, int16_t cmdDreapta)
30 {
31     if (cmdStanga > 255) cmdStanga = 255;
32     if (cmdStanga < -255) cmdStanga = -255;
33     if (cmdDreapta > 255) cmdDreapta = 255;
34     if (cmdDreapta < -255) cmdDreapta = -255;
35
36     cmdStangaAplicata = aplicaRampa(cmdStangaAplicata, cmdStanga
   ↪ );
37     cmdDreaptaAplicata = aplicaRampa(cmdDreaptaAplicata,
   ↪ cmdDreapta);
38
39     if (cmdStangaAplicata >= 0) {
40         PORTD |= (1 << PD4);
41         sensStanga = +1; /* citit de ISR-ul encoderului
   ↪ */
42     } else {
43         PORTD &= ~(1 << PD4);
44         sensStanga = -1;
45     }
46
47     if (cmdDreaptaAplicata >= 0) {
48         PORTD |= (1 << PD7);
49         sensDreapta = +1;
50     } else {
51         PORTD &= ~(1 << PD7);
52         sensDreapta = -1;
53     }
54
55     OCR1A = pwmDinComanda(cmdStangaAplicata < 0 ? -
   ↪ cmdStangaAplicata
56
   :
```

```

57         OCR1B = pwmDinComanda(cmdDreaptaAplicata < 0 ? -
        ↪ cmdDreaptaAplicata
58                                     :
        ↪ cmdDreaptaAplicata
        ↪ );
59 }
60
61 /* Opreire de urgenta: ocoleste DELIBERAT rampa, pentru a respecta
    ↪ cerinta
62    de oprire in cel mult 50 ms din orice stare. */
63 void motoareOpreireImediata(void)
64 {
65     OCR1A = 0;
66     OCR1B = 0;
67     cmdStangaAplicata = 0;
68     cmdDreaptaAplicata = 0;
69 }

```

11.7 Odometria din encodere

Fiecare roată motoare poartă un disc cu fante, citit de o pereche formată dintr-un LED și un fototranzistor. La fiecare fantă care trece prin fascicul, ieșirea comparatorului produce un front, iar numărarea fronturilor dă drumul parcurs de roată. Cu un disc de 60 de fante și o roată de 80 mm diametru, un front corespunde, cum am calculat mai devreme, la 4,19 mm de drum; numărând ambele fronturi, rezoluția se dublează, la 2,09 mm.

Întrebarea de proiectare este unde se face numărarea. Răspunsul, categoric, este în rutina de întrerupere, și motivul este cantitativ. Prin scrutare, programul ar trebui să citească pinul mai des decât se poate produce un impuls, altfel pierde fronturi și distanța măsurată devine mai mică decât cea reală, cu o eroare care crește exact atunci când robotul merge repede, adică atunci când contează. La 0,6 m/s, impulsurile vin la circa 7 ms unul de altul pe fiecare roată; o buclă de 10 ms le-ar pierde pe majoritatea. A număra în întrerupere rezolvă problema complet, iar calculul din exemplul anterior a arătat că prețul este de circa o zecime de procent din timpul procesorului.

Simetric, calculul nu are ce căuta în întrerupere. Rutina trebuie să facă un singur lucru: să incrementeze sau să decrementeze un contor. Conversia în milimetri, împărțirea la intervalul de timp, calculul vitezei liniare și al celei unghiulare, toate acestea se fac o dată pe ciclu, în bucla principală, unde durata lor nu deranjează pe nimeni. O rutină de întrerupere care ar face o împărțire în virgulă mobilă ar dura sute de microsecunde, întârziind toate celelalte întreruperi, inclusiv pe cea a celeilalte roți.

Sensul de numărare pune o problemă interesantă. Un encoder cu un singur canal nu poate distinge sensul de rotație: fronturile arată la fel indiferent încotro se învârtă roata. Soluția riguroasă este encoderul în cuadratură, cu două canale decalate cu un sfert de perioadă, dar acesta ar fi cerut încă două intrări de întrerupere, pe care nu le avem. Soluția de compromis, folosită aici, este ca rutina să adune contorul cu semnul comenzii date motorului, presupunând că roata se învârtă încotro a fost trimisă. Presupunerea este falsă în două situații: în timpul frânării, când inerția duce roata mai departe câteva zeci de milisecunde, și atunci când o roată este blocată de un obstacol și împinsă înapoi. Ambele introduc erori de odometrie, iar echipa a trebuit să le accepte ca preț al economiei de pini.

Exemplu 11.3 Numararea impulsurilor de encoder in intrerupere si citirea coerenta din bucla

```

1  volatile int32_t nImpulsStanga  = 0;
2  volatile int32_t nImpulsDreapta = 0;
3
4  /* Sensul presupus al fiecărei roti , scris de motoareComanda() ,
   ↪ citit de ISR.
5   Fiind pe un singur octet , accesul este atomic pe AVR. */
6  volatile int8_t sensStanga  = +1;
7  volatile int8_t sensDreapta = +1;
8
9  ISR(INT0_vect)                                /* front crescator , encoderul
   ↪ stang */
10 {
11     nImpulsStanga += sensStanga;
12 }
13
14 ISR(INT1_vect)                                /* front crescator , encoderul
   ↪ drept */
15 {
16     nImpulsDreapta += sensDreapta;
17 }
18
19 /* Contorul are patru octeti , iar intreruperea il poate modifica
   ↪ intre doi
20  octeti cititi de bucla principala. Citirea se protejeaza , dar
   ↪ starea
21  anterioara a bitului I se reface , ca sa nu validam
   ↪ intreruperile intr-o
22  sectiune in care erau deja oprite. */
23 int32_t encoderCiteste(volatile int32_t *pContor)
24 {

```

```

25     int32_t v;
26     uint8_t sreg = SREG;
27
28     cli();
29     v = *pContor;
30     SREG = sreg;
31     return v;
32 }

```

Exemplu

Într-un ciclu al buclei de reglare, cu perioada de 10 ms, contoarele au crescut cu 12 impulsuri pe roata stângă și cu 10 pe cea dreaptă. Ecartamentul robotului, adică distanța dintre punctele de contact ale celor două roți, este de 150 mm. Se cere viteza de deplasare și viteza de rotație a robotului, precum și eroarea introdusă de cuantizarea numărării.

Rezolvare. Un impuls înseamnă 4,19 mm, deci roata stângă a parcurs $12 \cdot 4,19 = 50,3$ mm, iar cea dreaptă $10 \cdot 4,19 = 41,9$ mm. Împărțind la 0,01 s se obțin vitezele $v_s = 5030$ mm/s și $v_d = 4190$ mm/s. Aceste valori sunt evident absurde pentru un robot mic, ceea ce arată că fereastra de măsurare a fost aleasă greșit: în 10 ms, la 0,5 m/s, sosesc doar circa un impuls pe roată, iar rezultatul este dominat de cuantizare. Refăcând calculul pe o fereastră de 100 ms, cu aceleași 12 și 10 impulsuri, se obține $v_s = 503$ mm/s și $v_d = 419$ mm/s, deci viteza de înaintare $v = (v_s + v_d)/2 = 461$ mm/s și viteza unghiulară $\omega = (v_d - v_s)/B = (419 - 503)/150 = -0,56$ rad/s, adică o rotire spre dreapta, ceea ce se potrivește cu faptul că roata dreaptă merge mai încet.

Cuantizarea introduce o incertitudine de un impuls, adică $4,19/0,1 = 41,9$ mm/s, deci circa opt la sută din viteză. Concluzia practică este că fereastra de calcul a vitezei trebuie să fie mai lungă decât ciclul de reglare: contoarele se citesc la fiecare 10 ms, dar viteza se calculează pe ultimele zece citiri. Se plătește o întârziere de 100 ms în estimarea vitezei, ceea ce este acceptabil, pentru că viteza nu este folosită la urmărirea liniei, ci la măsurarea drumului și la raportare.

11.8 Urmărirea liniei

Cei patru senzori de linie sunt perechi de LED și fototranzistor îndreptate spre podea, montate transversal în fața robotului. Suprafața deschisă reflectă mult, banda închisă reflectă puțin, iar fototranzistorul transformă diferența într-o tensiune pe care o citim pe un canal analogic. Prima operație obligatorie este *normalizarea*: pentru fiecare senzor se memorează, într-o procedură de calibrare făcută pe pistă înainte de probă, valoarea brută pe fond deschis și valoarea brută pe bandă, iar în funcționare valoarea citită se aduce, prin interpolare liniară, la o scară comună de la 0 la 100. Fără această operație, diferențele dintre exemplarele de fototranzistoare și dintre înălțimile de montaj ar face ca senzorii să nu fie comparabili între ei, iar poziția calculată ar fi deplasată sistematic.

Poziția liniei sub robot se obține ca o medie ponderată. Fiecărui senzor i se atribuie o poziție convențională, aici -3 , -1 , $+1$ și $+3$, măsurată în jumătăți de distanță dintre senzori, iar eroarea este centrul de greutate al răspunsurilor normalizate. Suma răspunsurilor are, în plus, o utilizare proprie: dacă este mai mică decât un prag, niciun senzor nu vede banda, deci linia este pierdută, iar eroarea nu are sens și nu trebuie folosită. Această verificare este implementarea concretă a cerinței CN4.

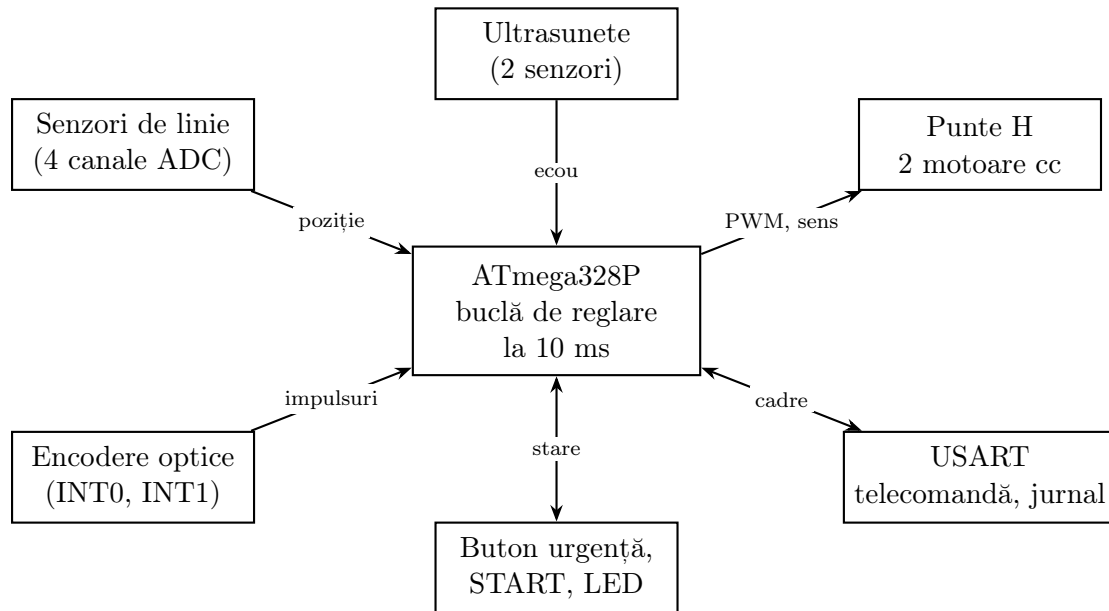


Figura 11.3 Bucla de reglare a urmăririi liniei. Reacția nu se închide printr-un fir, ci prin mișcarea fizică a robotului.

Legea de comandă, arătată ca schemă în Figura 11.3, este cea mai simplă cu putință: corecția este proporțională cu eroarea, iar aceeași corecție se adună la comanda unei roți și se scade din celelalte. Robotul înaintează astfel cu o viteză de bază și se rotește cu atât mai puternic cu cât este mai deplasat față de linie. Coeficientul K_p este singurul parametru de acordat, iar comportarea lui este ușor de observat pe teren: cu K_p prea mic, robotul reacționează leneș, taie curbele pe interior și pierde banda la prima curbă strânsă; cu K_p prea mare, robotul oscilează în jurul liniei cu amplitudine crescândă, pentru că fiecare corecție îl duce dincolo de traseu, iar corecția următoare, la fel de puternică, îl aduce înapoi și mai departe. Limita superioară a lui K_p este dată de întârzierea totală din buclă, formată din perioada ciclului, din constanta de timp a motoarelor și din inerția robotului; cu cât această întârziere este mai mare, cu atât K_p trebuie să fie mai mic.

Exemplu 11.4 Calculul erorii de pozitie si regulatorul proportional

```

1 #define KP_LINIE      18    /* unitati de comanda pe unitate de
   ↪ eroare */
2 #define VITEZA_BAZA   120
3 #define PRAG_LINIE    40    /* suma normalizata sub care linia e
   ↪ pierduta */

```

```

4
5 static const int8_t pozitie[4] = { -3, -1, +1, +3 };
6
7 /* n[i]: raspunsul normalizat al senzorului i, 0..100.
8    Returneaza eroarea in sutimi de unitate de pozitie.
9    *pSuma primeste suma raspunsurilor, folosita la detectia liniei
        ↪ pierdute. */
10 int16_t linieEroare(const uint8_t n[4], int16_t *pSuma)
11 {
12     int16_t suma = 0;
13     int32_t moment = 0;
14     uint8_t i;
15
16     for (i = 0; i < 4; i++) {
17         suma += n[i];
18         moment += (int32_t)pozitie[i] * n[i];
19     }
20     *pSuma = suma;
21
22     if (suma < PRAG_LINIE) {
23         return 0;                                /* eroare nedefinita: linia e
        ↪ pierduta */
24     }
25     return (int16_t)(moment * 100 / suma);
26 }
27
28 void regulatorLinie(void)
29 {
30     int16_t suma;
31     int16_t e = linieEroare(nSenzor, &suma);
32     int16_t u = (int16_t)((int32_t)KP_LINIE * e) / 100;
33
34     motoareComanda(VITEZA_BAZA + u, VITEZA_BAZA - u);
35 }

```

Exemplu

Senzorii de linie, normalizați pe scara de la 0 la 100, indică valorile 5, 20, 90 și 35, de la stânga la dreapta. Viteza de bază este 120, iar $K_p = 18$. Se cere eroarea de poziție și comenzile trimise celor două motoare.

Rezolvare. Suma răspunsurilor este $5 + 20 + 90 + 35 = 150$, mult peste pragul de 40,

deci linia este vizibilă. Momentul este $(-3) \cdot 5 + (-1) \cdot 20 + (+1) \cdot 90 + (+3) \cdot 35 = -15 - 20 + 90 + 105 = 160$. Eroarea, exprimată în sutimi, este $160 \cdot 100/150 = 106$, adică 1,06 unități de poziție spre dreapta: banda se află la dreapta axului robotului. Corecția este $u = 18 \cdot 106/100 = 19$. Comenzile devin $120 + 19 = 139$ pentru motorul stâng și $120 - 19 = 101$ pentru cel drept, deci robotul virează spre dreapta, către bandă, exact cum trebuie.

Se poate estima și corecția maximă posibilă: eroarea nu poate depăși ± 3 unități, adică ± 300 în sutimi, deci u rămâne între -54 și $+54$. Cu viteza de bază 120, comenzile rămân între 66 și 174, deci nu se saturează niciodată la 255. Aceasta este o proprietate bună: dacă la eroare maximă comenzile s-ar satura, robotul ar pierde capacitatea de a mai corecta suplimentar și s-ar comporta neliniar tocmai în situația cea mai grea. Observăm de asemenea că toate calculele s-au făcut cu numere întregi, cu factorul de scară 100 păstrat explicit; pe un microcontroler fără unitate în virgulă mobilă, aceasta nu este o preferință de stil, ci diferența dintre câteva microsecunde și câteva sute.

11.9 Planul de testare

Planul de testare al robotului urmează cele patru niveluri descrise în capitolul precedent, dar are o particularitate: ultimul nivel nu se poate face în laborator. Un robot care merge impecabil pe masa de lucru poate eșua pe pista de concurs din motive care nu au nimic de-a face cu programul, precum iluminarea sălii sau aderența pardoselii. De aceea planul prevede explicit un nivel de teren, cu criterii proprii.

Nivelul de modul verifică funcțiile pur logice, fără hardware. Funcția de calcul al erorii, funcția de compresie a comenzii peste zona moartă și funcția de rampă sunt compilate pentru calculatorul gazdă și încercate cu valori la limite: toți senzorii pe zero, un singur senzor activ la extremitate, sume exact la prag. Un asemenea test costă câteva minute și prinde clasa de erori cea mai frecventă, aceea a comparațiilor greșite la capete de domeniu.

Nivelul de subsistem verifică un modul întreg, cu hardware, dar izolat de restul. Roțile se ridică de pe sol și se comandă motoarele cu valori crescătoare, notând valoarea la care fiecare roată începe să se rotească; de aici rezultă pragul zonei moarte. Se rotește manual fiecare roată exact zece ture și se citește contorul encoderului, care trebuie să indice 600 de impulsuri; orice abatere arată fie fronturi pierdute, fie fronturi parazite din cauza reboundului optic la trecerea prin dreptul fantei. Senzorul cu ultrasunete se verifică față de o suprafață plană așezată la distanțe cunoscute cu ruleta.

Nivelul de integrare verifică interacțiunile, adică exact ceea ce nivelurile anterioare nu pot verifica. Aici apar defectele interesante: dacă motoarele pornesc, mai numără corect encoderele, sau perturbațiile electrice ale punții produc fronturi false? Dacă jurnalul serial transmite continuu, mai respectă bucla perioada de 10 ms? Dacă senzorul cu ultrasunete măsoară în același timp cu conversiile analogice, mai sunt acestea corecte?

Nivelul de teren rulează scenariile complete pe pistă: tur complet, obstacol apărut în curbă, bandă întreruptă, oprire de urgență în plină viteză, acumulator descărcat pe parcurs.

Tabelul 11.4 arată un fragment, cu legătura explicită către cerințe.

Tabela 11.4 Fragment din planul de testare al robotului, pe niveluri.

Test	Cerință	Mod de efectuare	Criteriu de acceptare
T1	CN4	Se apelează funcția de eroare cu toate valorile zero	Returnează suma sub prag, fără împărțire la zero
T2	-	Roți ridicate, comandă crescătoare unitate cu unitate	Pragul zonei moarte este identic la ambele roți, ± 5 unități
T3	CF7	Zece rotații manuale ale fiecărei roți	Contorul indică 600 ± 2 impulsuri
T4	CF3	Panou plan la 100, 250 și 500 mm, câte 20 de măsurări	Eroare sub 10 mm, fără citiri lipsă
T5	CN1	Pin de test comutat la intrarea și ieșirea din prelucrare	Impuls sub 5 ms, perioada 10 ± 1 ms
T6	-	Motoare la comanda maximă, roțile blocate	Contoarele encoderelor nu cresc cu fronturi parazite
T7	CF1	Zece tururi complete pe pista de antrenament	Nicio pierdere a benzii, niciun contact cu marginea
T8	CF5	Buton de urgență apăsat la viteză maximă, 20 de repetări	Comanda punții la zero în sub 50 ms, măsurat cu osciloscopul
T9	CF4	Robotul este ridicat de pe pistă în timpul deplasării	Oprire după 2 s, semnalizare, rămâne oprit
T10	CN3	Alimentare de la sursă reglabilă, coborâre lentă	Oprire controlată la pragul stabilit, mesaj în jurnal
T11	-	Comenzi eronate și caractere aleatoare pe interfața serială	Cadrelle invalide sunt ignorate, robotul nu se blochează
T12	-	Pornirea de o sută de ori cu butonul de start ținut apăsat	Sistemul rămâne în stare definită, fără porniri repetate

Testele T2, T6, T11 și T12 nu au o cerință asociată, ceea ce înseamnă, ca și în capitolul precedent, că ele provin din testarea atipică. Dacă un asemenea test descoperă un defect, corectura se face nu numai în cod, ci și în caietul de sarcini, care se completează cu cerința care lipsea. Testul T6 în special merită atenție: el verifică imunitatea la perturbații, iar dacă eșuează, remediul nu este în software, ci în cablaj, prin separarea traseelor de forță de cele de semnal, prin torsadarea firelor de motor și prin condensatoare de decuplare la bornele motorului.

11.10 Ce a mers și ce nu

Un studiu de caz care raportează numai succese nu este de niciun folos. Cele mai utile rezultate ale unui proiect de acest fel sunt greșelile, pentru că ele se repetă cu o regularitate remarcabilă la fiecare generație de echipe.

A mers bine ceea ce a fost decis pe hârtie înainte de a fi construit. Partiționarea pe subsisteme cu interfețe scrise a permis ca patru persoane să lucreze în paralel și ca modulul de tracțiune să fie înlocuit, la jumătatea proiectului, cu unul construit pe alt circuit de comandă, fără nicio modificare în restul programului. Mașina cu stări a făcut ca situațiile neprevăzute să aibă totuși un comportament definit. Regula de a nu pune nimic lent în bucla principală a fost respectată de la început și nu a trebuit niciodată reconsiderată, iar rezerva de timp măsurată a permis adăugarea odometriei în ultima săptămână.

Nu a mers ceea ce a fost decis din mers. Prima greșală a fost calibrarea senzorilor de linie cu valori compilate în program: la concurs, iluminarea sălii și nuanța pardoselii au fost altele decât în laborator, iar robotul nu a mai văzut banda. Remedierea, făcută în grabă, a fost o procedură de calibrare pornită prin butonul de start, care memorează valorile extreme în timp ce robotul este plimbat manual peste bandă. Lecția generală este că orice parametru care depinde de mediu trebuie măsurat la fața locului, nu presupus.

A doua greșală a fost alimentarea comună a electronicii și a motoarelor de la același acumulator, fără separare. La accelerarea bruscă, tensiunea cădea suficient pentru ca microcontrolerul să se reseteze, iar robotul repornea în starea de așteptare, în mijlocul pistei. Defectul a apărut abia pe pista reală, pentru că în laborator alimentarea se făcea de la o sursă de rețea, capabilă să susțină vârful de curent. Remedierile au fost trei, aplicate împreună: un condensator de capacitate mare la intrarea stabilizatorului electronicii, rampa de accelerare descrisă mai sus și un circuit de supraveghere a tensiunii de alimentare, care ține microcontrolerul în resetare până când tensiunea este sigură.

A treia greșală a fost încrederea în odometrie ca sursă de poziție absolută. Erorile de cuantizare, patinarea roților și, mai ales, sensul presupus în locul celui măsurat se acumulează, iar după câțiva metri poziția estimată se depărtează vizibil de cea reală. Odometria este utilă pentru manevre scurte, precum ocolirea unui obstacol, unde eroarea nu are timp să se adune, dar nu este o soluție de navigație pe termen lung fără o corecție periodică dintr-o măsurătoare independentă, cum ar fi trecerea peste un marcaj cunoscut.

Observație

Toate cele trei greșeli au un element comun: au fost descoperite la trecerea de la mediul de laborator la mediul real, adică exact la nivelul de testare pe care echipele îl programează ultimul și îl scurtează primul atunci când timpul se termină. Concluzia practică este să se meargă pe pista reală cât mai devreme, cu un robot incomplet, în loc să se meargă târziu cu unul terminat.

Rămâne o observație finală despre metodă. Nimic din ceea ce a făcut acest proiect posibil nu a fost o cunoștință nouă despre ATmega328P: registrele timerelor, biții de configurare

ai întreruperilor și lanțul de conversie al convertorului analog-numeric erau toate cunoscute din capitolele anterioare. Ceea ce a decis rezultatul a fost ordinea în care au fost luate deciziile, disciplina de a scrie interfețele înaintea implementărilor și obiceiul de a verifica prin măsurare, nu prin presupunere. Aceasta este, în esență, diferența dintre a ști să programezi un microcontroler și a ști să proiectezi un sistem cu microcontroler.

Întrebări de verificare

1. Regulamentul unui concurs cere ca robotul „să evite coliziunile”. Transformați această formulare în două sau trei cerințe verificabile, cu valori numerice și cu metoda de măsurare corespunzătoare fiecăreia.
2. Explicați de ce butonul de oprire de urgență nu trebuie să acționeze numai prin software. Descrieți o schemă în care butonul întrerupe fizic alimentarea etajului de putere și anunță totodată microcontrolerul.
3. Refaceți harta pinilor din acest capitol pentru un robot cu șase senzori de linie și trei senzori cu ultrasunete. Arătați ce resurse lipsesc pe ATmega328P și propuneți două soluții diferite, indicând prețul fiecăreia.
4. De ce nu a putut fi folosit modulul de captură al lui Timer1 pentru măsurarea ecoului, deși aceasta este metoda recomandată? Ce configurație a lui Timer1 ar face captura utilizabilă și ce s-ar pierde în schimb?
5. Explicați ce se întâmplă cu bucla principală dacă măsurarea ecoului ultrasonic se face prin așteptarea activă a frontului. Estimați numeric întârzierea maximă introdusă și arătați care cerință din caietul de sarcini ar fi încălcată.
6. Un coleg mută calculul vitezelor din bucla principală în rutina de întrerupere a encoderului, „ca să fie mai actual”. Explicați ce se strică, inclusiv efectul asupra celeilalte roți și asupra măsurării ecourilor.
7. Ce este zona moartă a unui motor cu reductor, cum se determină experimental pragul ei și ce comportament se observă la un robot care urmărește o linie fără compensarea acestei zone?
8. Un encoder cu un singur canal nu poate determina sensul de rotație. Descrieți două situații în care presupunerea că roata se rotește în sensul comandat este falsă și estimați eroarea de odometrie introdusă în fiecare caz.
9. Pentru robotul din acest capitol, calculați rezoluția odometriei dacă discul are 30 de fante în loc de 60, iar numărarea se face pe ambele fronturi. Ce fereastră de timp este necesară pentru ca eroarea de cuantizare a vitezei să rămână sub cinci la sută la 0,5 m/s?

10. Arătați ce se întâmplă cu robotul dacă K_p este dublat față de valoarea din exemplu. Calculați corecția maximă și precizați de la ce valoare a lui K_p comenzile încep să se satureze, la aceeași viteză de bază.
11. Explicați de ce testele T2, T6, T11 și T12 din planul de testare nu au o cerință asociată și ce trebuie făcut dacă unul dintre ele eșuează. Propuneți cerințele care ar trebui adăugate în caietul de sarcini.
12. Descrieți procedura de calibrare a senzorilor de linie pe pista de concurs și explicați de ce valorile compilate în program nu pot înlocui această procedură.