

Programare embedded - lucrul cu biți și regiștri

Durată: 2 ore

Tip: lucrare pregătitoare

Necesar pentru: Lab. 2-7

Platformă: orice microcontroler

CUPRINS

1. Obiectivele lucrării
2. De ce lucrăm pe regiștri
3. Baze de numerație
4. Operatorii pe biți
5. Laborator de operatori
6. Construirea măștilor
7. Simulator de măști
8. Cele patru idiomuri
9. Câmpuri de mai mulți biți
10. Citește-modifică-scrie
11. Atomicitate
12. Simulator: coruperea RMW
13. Regiștri mapați în memorie
14. Portarea către alte controlere
15. Cod sursă
16. Montaj interactiv
17. Sarcini de lucru
18. Provocare
19. Verificare
20. Resurse

Această lucrare pregătitoare fixează singura tehnică fără de care restul semestrului nu are sens: manipularea individuală a biților dintr-un registru. Funcțiile de bibliotecă precum `digitalWrite()` există doar pe Arduino; măștile de biți funcționează identic pe AVR, STM32, ESP32, PIC sau orice alt microcontroler din lume.

LUCRARE SUPLIMENTARĂ

Această ședință nu are corespondent în îndrumarul tipărit. Ea acoperă noțiunile presupuse cunoscute de laboratoarele următoare și se recomandă parcurgerea ei înaintea Laboratorului 2.

1 Obiectivele lucrării

- ☐ Conversia rapidă între binar, hexazecimal și zecimal
- ☐ Utilizarea corectă a celor șase operatori pe biți
- ☐ Construirea măștilor cu operatorul de deplasare
- ☐ Aplicarea celor patru idiomuri fundamentale: setare, ștergere, comutare, testare
- ☐ Scrierea și citirea câmpurilor de mai mulți biți
- ☐ Înțelegerea naturii citește–modifică–scrie și a problemelor de atomicitate
- ☐ Scrierea de cod portabil, independent de biblioteca Arduino

2 De ce lucrăm direct pe regiștri

Un microcontroler se configurează exclusiv prin scrierea unor valori în regiștri - locații speciale de memorie legate direct la hardware. Fiecare bit dintr-un registru comandă o funcție concretă: direcția unui pin, activarea unui periferic, alegerea unui prescaler.

Aspect	Funcție de bibliotecă	Acces direct la registru
Viteză	~50 cicluri ($\approx 3 \mu s$ la 16 MHz)	1 ciclu (62,5 ns)
Memorie Flash ocupată	sute de octeți de cod auxiliar	o instrucțiune
Portabilitate	doar unde există biblioteca	oriunde, cu alte nume de regiștri
Control	doar ce a prevăzut autorul bibliotecii	tot ce permite hardware-ul
Lizibilitate	foarte bună pentru începători	necesită documentația (datasheet)
Depanare	ascunde ce se întâmplă	corespondență directă cu datasheet-ul

POZIȚIA ACESTUI CURS

Vom folosi în continuare și funcțiile Arduino, pentru că scurtează prototiparea. Dar fiecare noțiune va fi prezentată

ȘI

la nivel de registru: doar așa veți putea citi un datasheet, folosi un periferic pe care biblioteca nu îl expune și trece de la Arduino la STM32 sau ESP32 fără să reînvățați totul.

FUNCȚIILE DE BIBLIOTECĂ NU ACOPERĂ TOT

Nu există o funcție gata făcută care să configureze un numărător hardware să comande singur un pin, sau care să pornească o conversie analog-numerică declanșată de un comparator - lucruri pe care le veți face la lucrările următoare, scriind direct în regiștri. Din momentul în care aplicația depășește nivelul de demonstrație, accesul la regiștri devine obligatoriu.

3 Baze de numerație

Un registru de 8 biți se scrie natural în binar, dar binarul este greu de citit. Hexazecimalul este compromisul standard: fiecare cifră hexazecimală corespunde exact la **4 biți**.

Binar	Hexa	Zecimal	Binar	Hexa	Zecimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

aceeași valoare, trei notații

```
0b10110100    // binar   - se vede fiecare bit
0xB4           // hexa    - compact: B=1011, 4=0100
180            // zecimal - nu spune nimic despre biti

// toate trei sunt IDENTICE pentru compilator
```

CONVERSIE RAPIDĂ BINAR → HEXA

Se grupează biții câte patru, de la dreapta: 1011 0100 → B și 4 → 0xB4. Invers la fel de simplu. Cu exercițiu, conversia devine instantanee - și veți avea nevoie de ea la fiecare pagină de datasheet.

Poziția bitului	7	6	5	4	3	2	1	0
-----------------	---	---	---	---	---	---	---	---

Valoare zecimală	128	64	32	16	8	4	2	1
<code>1 << n</code>	0x80	0x40	0x20	0x10	0x08	0x04	0x02	0x01

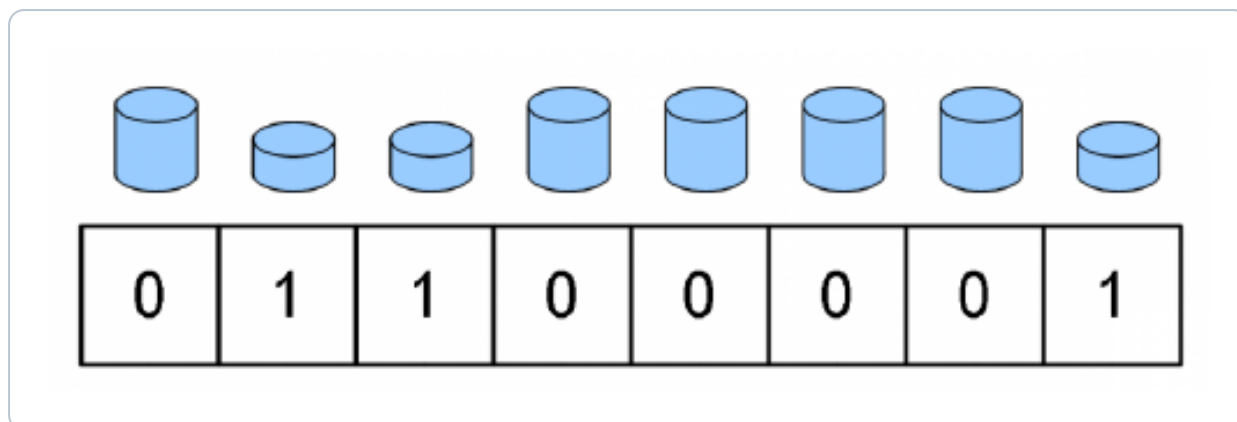


Fig. 1 - Un registru de 8 biți: opt celule, fiecare păstrând 0 sau 1. Valoarea din figură este `0b01100001 = 0x61 = 97`. [Fig. 2.3 din îndrumar](#)

4 Operatorii pe biți

Operator	Nume	Regulă	Folosit pentru
<code>&</code>	ȘI	1 doar dacă ambii biți sunt 1	testarea și ștergerea biților
<code> </code>	SAU	1 dacă cel puțin un bit este 1	setarea biților
<code>^</code>	SAU exclusiv	1 dacă biții diferă	comutarea biților
<code>~</code>	negare	inversează fiecare bit	construirea măștilor de ștergere
<code><<</code>	deplasare stânga	mută biții spre stânga, completează cu 0	poziționarea măștii
<code>>></code>	deplasare dreapta	mută biții spre dreapta	extragerea câmpurilor

Tabelele de adevăr

A	B	A & B	A B	A ^ B	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

& NU ESTE &&

1 & 2 dă

0

- biții nu se suprapun (01 și 10). Dar 1 && 2 dă

1

- ambele valori sunt nenule, deci adevărate. Aceeași capcană există între | și ||. Confuzia produce erori care se compilează fără nicio avertizare.

Proprietățile care fac totul să funcționeze

Proprietate	Consecință practică
$x 0 = x$	SAU cu 0 nu schimbă bitul → biții din mască puși pe 0 rămân neatinși
$x 1 = 1$	SAU cu 1 forțează pe 1 → biții din mască puși pe 1 se setează
$x \& 1 = x$	ȘI cu 1 nu schimbă bitul → biții din mască puși pe 1 rămân neatinși
$x \& 0 = 0$	ȘI cu 0 forțează pe 0 → biții din mască puși pe 0 se șterg
$x \wedge 0 = x$	XOR cu 0 nu schimbă bitul
$x \wedge 1 = \sim x$	XOR cu 1 inversează bitul

ACESTA ESTE ÎNTREGUL SECRET

Toate idiomurile de mai jos decurg din tabelul acesta. Dacă îl înțelegeți, nu mai aveți nimic de memorat.

5 Laborator de operatori

Comutați biții direct pe reprezentarea binară și urmăriți cum se formează rezultatul, poziție cu poziție. Verificați experimental fiecare proprietate din tabelul de mai sus.

► OPERATORI PE BIȚI - BIT CU BIT

OPERAND A

60

OPERATOR

OPERAND B / POZIȚII

13

A = 60

7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	0

B = 13

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1

Rezultat = 12

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0

```
expresie: 60 & 13 = 12
binar: 0b00001100
hexazecimal: 0x0C
```

Apăsați pe biți pentru a-i comuta. Operatorii pe biți lucrează pe fiecare poziție independent.

[Versiune interactivă pe site.](#)

6 Construirea măștilor

O **mască** este o valoare în care biții care ne interesează sunt marcați cu 1. Se construiește cu operatorul de deplasare, pornind de la valoarea 1:

cum se deplasează un bit

```
1          -> 0000 0001    (bitul 0)
1 << 1     -> 0000 0010    (bitul 1)
1 << 2     -> 0000 0100    (bitul 2)
1 << 5     -> 0010 0000    (bitul 5)
1 << 7     -> 1000 0000    (bitul 7)

~(1 << 5)  -> 1101 1111    (toti biti 1, mai putin bitul 5)

(1 << 3) | (1 << 1) -> 0000 1010 (bitii 3 SI 1 deodata)
```

DE CE NU SCRIM DIRECT VALOAREA

`PORTB |= 0b00100000;` funcționează, dar trebuie să numărăm pozițiile de fiecare dată. `PORTB |= (1 << PB5);` spune explicit **CARE**

bit, folosind numele din datasheet. Compilatorul produce exact același cod mașină - deplasarea se calculează la compilare, nu la execuție.

ATENȚIE LA TIPUL REZULTATULUI

Pe AVR, 1 este un `int` de 16 biți. Pentru regiștri pe 8 biți nu contează, dar pentru bitul 31 al unui registru pe 32 de biți (STM32, ESP32) trebuie scris `1UL << 31`, altfel rezultatul este nedefinit.

7 Simulator de măști

Alegeți o operație și o poziție de bit, apoi apăsați „aplică”. Urmăriți masca generată, expresia corespunzătoare și bitul care se modifică - evidențiat cu portocaliu. Observați că restul biților rămân întotdeauna neatinși.

► CONSTRUIREA ȘI APLICAREA MĂȘTII PE UN REGISTRU

OPERAȚIE POZIȚIA BITULUI (N)

PORTB - valoare curentă

7 PB7	6 PB6	5 PB5	4 PB4	3 PB3	2 PB2	1 PB1	0 PB0
0	0	0	0	1	0	1	0

masca (1 << 5)

7 PB7	6 PB6	5 PB5	4 PB4	3 PB3	2 PB2	1 PB1	0 PB0
0	0	1	0	0	0	0	0

expresie: **PORTB |= (1 << 5);**
 masca: **0b00100000** = 0x20
 PORTB acum: **0b00001010** = 0x0A = 10
 SAU pe biți: bitul 5 devine 1, ceilalți rămân neschimbați.

Toate cele trei operații de scriere sunt de tip **citește-modifică-scrie**: microcontrolerul citește registrul, calculează noua valoare și o scrie înapoi. Sunt trei instrucțiuni, nu una.

Versiune interactivă pe site.

DE ÎNCERCAT

Porniți de la valoarea inițială 0b00001010. Setati bitul 5, apoi ștergeți bitul 1, apoi comutați bitul 3 de două ori. Verificați de fiecare dată că biții neimplicați și-au păstrat valoarea - aceasta este întreaga rațiune a măștilor.

8 Cele patru idiomuri fundamentale

Practic tot codul embedded din lume folosește aceste patru forme. Merită memorate ca atare:

Operație	Idiom	De ce funcționează
Setare (pune pe 1)	<code>REG = (1 << n);</code>	SAU cu 1 forțează 1; SAU cu 0 nu schimbă nimic
Ștergere (pune pe 0)	<code>REG &= ~(1 << n);</code>	ȘI cu 0 forțează 0; ȘI cu 1 nu schimbă nimic
Comutare	<code>REG ^= (1 << n);</code>	XOR cu 1 inversează; XOR cu 0 nu schimbă nimic
Testare	<code>if (REG & (1 << n))</code>	ȘI izolează bitul; rezultat nenul dacă bitul e 1

macrouri portabile - de folosit în toate lucrările

```
#define BIT(n)                (1UL << (n))

#define SET_BIT(reg, n)       ((reg) |=  BIT(n))
#define CLR_BIT(reg, n)       ((reg) &= ~BIT(n))
#define TGL_BIT(reg, n)       ((reg) ^=  BIT(n))
#define CHK_BIT(reg, n)       (((reg) &  BIT(n)) != 0)

// scrie o valoare de 0 sau 1 intr-un bit, fara ramificatie
#define WRT_BIT(reg, n, v)    ((reg) = ((reg) & ~BIT(n)) | ((v) ? BIT(n) : 0))

// exemple de utilizare
SET_BIT(DDRB,  PB5);          // pinul 13 devine iesire
SET_BIT(PORTB, PB5);          // pinul 13 pe HIGH
CLR_BIT(PORTB, PB5);          // pinul 13 pe LOW
TGL_BIT(PORTB, PB5);          // comuta pinul 13

if (CHK_BIT(PIND, PD2)) {    // butonul NU este apasat (pull-up)
    // ...
}
```

PARANTEZELE NU SUNT OPȚIONALE

Într-un macro, fiecare parametru trebuie încadrat în paranteze. Fără ele, SET_BIT(x, a + 1) s-ar expanda greșit, din cauza priorității operatorilor. Aceasta este o sursă clasică de erori foarte greu de depistat.

9 Câmpuri de mai mulți biți

Multe setări nu ocupă un singur bit, ci un grup. De exemplu, prescalerul unui timer este dat de trei biți CS12:CS10, iar canalul ADC de patru biți MUX3:MUX0.

scrierea unui câmp de mai mulți biți

```
// Sa zicem ca bitii 3, 4, 5 ai portului B formeaza un "camp" de 3 biti
// (o valoare de la 0 la 7 trimisa catre un circuit exterior), iar bitii
// 0, 1, 2 comanda cu totul altceva si NU trebuie atinsi.

// GRESIT: sterge tot restul registrului
PORTB = 0b00101000;          // campul are valoarea 5... dar bitii 0-2 s-au pierdut!

// CORECT: se sterge doar campul, apoi se scrie valoarea
PORTB &= ~(1 << PB5) | (1 << PB4) | (1 << PB3)); // 1. curata campul
PORTB |=  (1 << PB5);          // 2. scrie valoarea noua

// Forma generala, cu masca si deplasare
#define FIELD_SET(reg, mask, shift, val) \
    ((reg) = ((reg) & ~(mask) << (shift)) | (((val) & (mask)) << (shift)))

// exemplu: scrie valoarea 5 in campul de 3 biti care incepe la pozitia 3
FIELD_SET(PORTB, 0x07, 3, 5);
```

Citirea unui câmp

extragerea valorii

```
// Se izoleaza campul cu ȘI, apoi se aduce la pozitia 0 prin deplasare
uint8_t jumatateaDeJos = (PORTB & 0x0F);           // bitii 0-3, incep la pozitia 0

// camp care NU incepe de la pozitia 0: intai deplasare, apoi masca
uint8_t campul3biti     = (PORTB >> 3) & 0x07;     // bitii 3-5

uint8_t bitulDeSus      = (PORTB >> 5) & 0x01;     // un singur bit, pozitia 5

#define FIELD_GET(reg, mask, shift) (((reg) >> (shift)) & (mask))
```

ORDINEA CONTEAZĂ

La citire se deplasează

ÎNTÂI

și se maschează

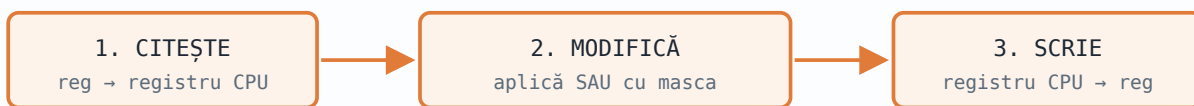
DUPĂ

. La scriere se maschează întâi valoarea (ca să nu deorbe peste câmpurile vecine) și abia apoi se deplasează. Inversarea ordinii produce valori greșite, fără nicio eroare de compilare.

10 Citește–modifică–scrie

Expresia `PORTB |= (1 << PB5);` arată ca o singură operație, dar procesorul o execută în **trei pași**:

`PORTB |= (1 << PB5);` → trei instrucțiuni distincte, nu una



o întrerupere care apare AICI vede o valoare intermediară, iar modificarea ei se va pierde

Fig. 1 - Structura reală a unei operații citește–modifică–scrie. Fereastra dintre pașii 2 și 3 este locul unde apar erorile de concurență.

ce generează compilatorul (AVR)

`PORTB |= (1 << PB5);`

; se traduce în:

```
in   r24, 0x05      ; 1. CITEȘTE PORTB în registrul r24
ori  r24, 0x20      ; 2. MODIFICĂ: SAU cu masca 0b00100000
out  0x05, r24      ; 3. SCRIE r24 înapoi în PORTB
```

EXCEPȚIE UTILĂ PE AVR

Pentru regiștrii aflați în primii 32 de octeți de I/O (printre care PORTB, PORTD, DDRB) și pentru o mască **CONSTANTĂ CU UN SINGUR BIT**

, compilatorul folosește instrucțiunile `sbi / cbi`, care sunt

ATOMICE

și durează un singur ciclu. Pentru regiștrii din memoria extinsă (TCCR1B, ADCSRA, UCSR0B) sau pentru măști calculate la execuție, se generează secvența completă de trei instrucțiuni.

11 Atomicitate

O operație este **atomică** dacă nu poate fi întreruptă la jumătate. Problema apare când același registru sau aceeași variabilă este modificată atât din programul principal, cât și dintr-o rutină de întrerupere.

Situație	Atomic?	Soluție
<code>PORTB = (1 << PB5);</code> - mască constantă, registru I/O jos	da (sbi)	nimic de făcut
<code>TCCR1B = (1 << CS12);</code> - registru extins	nu	secțiune critică, dacă ISR-ul atinge același registru
Citirea unui <code>uint16_t</code> modificat în ISR	nu	secțiune critică obligatorie
<code>contor++</code> pe <code>uint8_t</code> volatile	nu	secțiune critică
Citirea unui <code>uint8_t</code> volatile	da	doar <code>volatile</code>

secțiune critică - forma corectă

```
// GRESIT: reactiveaza intreruperile chiar daca apelantul le avea oprite
cli();
valoare = contor;
sei();                                // <-- presupune ca erau active. Nu e garantat!

// CORECT: se salveaza si se restaureaza starea exacta
uint8_t sreg = SREG;                 // SREG contine bitul I (Global Interrupt Enable)
cli();
valoare = contor;                     // sectiune critica: nimic nu o poate intrerupe
SREG = sreg;                          // restaureaza EXACT starea anterioara

// CEL MAI BINE: macro din biblioteca standard AVR
#include <util/atomic.h>

ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
    valoare = contor;                 // salvarea si restaurarea se fac automat
}
```

SECȚIUNILE CRITICE TREBUIE SĂ FIE FOARTE SCURTE

Cât timp întreruperile sunt dezactivate, sistemul este surd: se pot pierde fronturi, octeți primiți pe serial sau tick-uri de timer. Copiați valoarea în interiorul secțiunii și

PRELUCRAȚI-O ÎN AFARA EI**ALTERNATIVA FĂRĂ DEZACTIVAREA ÎNTRERUPERILOR**

Pentru un singur indicator boolean sau un contor pe 8 biți, volatile este suficient. Pentru structuri mai complexe se poate folosi tehnica „double read”: se citește variabila de două ori și se acceptă rezultatul doar dacă cele două citiri coincid.

12 Simulator: coruperea unei operații RMW

Urmăriți exact cum o întrerupere care apare între citire și scriere face ca modificarea ei să dispară fără urmă. Este una dintre cele mai greu de depistat erori din programarea embedded, pentru că apare rar și aparent aleator.

► CE SE ÎNTÂMPLĂ CÂND ISR-UL INTERVINE LA MIJLOC

```

1 // programul principal vrea sa SETEZE bitul 5
2 TCCR1B |= (1 << 5);
3
4 // ISR-ul vrea sa STEARGA bitul 0
5 ISR(TIMER0_OVF_vect) {
6     TCCR1B &= ~(1 << 0);
7 }
```

variabilă	valoare
TCCR1B (real)	0000 0001
r24 (CPU)	-
pas	înainte de operație

IEȘIRE



pasul 1 din 7 - registrul are bitul 0 setat

Versiune interactivă pe site.

CUM SE EVITĂ

Dacă un registru este atins atât din `loop()`, cât și dintr-un ISR, orice operație RMW din programul principal trebuie protejată cu `ATOMIC_BLOCK`. Alternativ - și de preferat - se stabilește regula ca fiecare registru să fie modificat dintr-un singur loc.

13 Regiștri mapați în memorie

Numele `PORTB` nu este o variabilă obișnuită: este o macrodefiniție care indică spre o adresă fizică din spațiul de memorie, legată direct la hardware.

ce se ascunde în spatele numelui

```
// In fisierele antet ale AVR:
#define PORTB (*(volatile uint8_t *)0x25)
//          |          |          |
//          |          |          | adresa fizica a registrului
//          |          |          | conversie la pointer spre octet volatil
//          |          |          | dereferentiere: lucram cu VALOAREA de la acea adresa

// Deci PORTB |= 0x20; inseamna de fapt:
*(volatile uint8_t *)0x25 |= 0x20;

// Acelasi lucru se poate scrie explicit:
volatile uint8_t *portb = (volatile uint8_t *)0x25;
*portb |= (1 << 5);
```

DE CE VOLATILE ESTE OBLIGATORIU AICI

Valoarea unui registru se poate schimba

FĂRĂ CA PROGRAMUL SĂ O MODIFICE

- de exemplu `PIND` reflectă starea fizică a pinilor. Fără volatile, compilatorul ar citi registrul o singură dată și ar refolosi valoarea memorată, iar bucla `while (!(UCSR0A & (1 << RXC0)))`; nu s-ar termina niciodată.

DE CE CONTEAZĂ

Odată înțeles acest mecanism, puteți accesa

ORICE

periferic de pe

ORICE

microcontroler, chiar dacă nu există bibliotecă pentru el: căutați adresa în datasheet și scrieți acolo. Aceasta este definiția programării bare-metal.

Bit	7	6	5	4	3	2	1	0	
0x0B (0x2B)	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	PORTD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
0x0A (0x2A)	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	DDRD
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
0x09 (0x29)	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	PIND
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Fig. 2 - Cei trei regiștri ai portului D, așa cum apar în datasheet: numele fiecărui bit, adresa în spațiul de I/O, drepturile de acces (R/W sau doar R) și valoarea de după reset. Observați că PIND este marcat R - doar citire. Fig. 2.4 din îndrumar

CUM SE CITEȘTE UN ASTFEL DE TABEL

Rândul

BIT

dă poziția (7 ... 0), rândul următor numele fiecărui bit - exact numele pe care îl folosiți în (1 << DDD4).

READ/WRITE

spune dacă puteți scrie în el, iar

INITIAL VALUE

valoarea de după reset. Veți regăsi acest format la absolut toți regiștrii, în toate lucrările următoare.

14 Portarea către alte controlere

Idiomurile pe biți rămân **identice**. Se schimbă doar numele regiștrilor și, uneori, arhitectura perifericului. Iată aceeași operație - „aprinde un LED” - pe patru platforme:

același concept, patru microcontrolere

```
// ----- AVR / ATmega328P (Arduino Uno) -----
DDRB  |= (1 << PB5);           // pin ca iesire
PORTB |= (1 << PB5);           // pin pe HIGH
PORTB &= ~(1 << PB5);          // pin pe LOW

// ----- STM32 (seria F1/F4, registre directe) -----
RCC->APB2ENR |= (1 << 3);       // porneste ceasul portului C
GPIOC->CRH   |= (1 << 20);      // configureaza pinul 13 ca iesire
GPIOC->BSRR   = (1 << 13);      // pin pe HIGH (registru dedicat, ATOMIC)
GPIOC->BSRR   = (1 << 29);      // pin pe LOW (bitul 13 + 16)

// ----- ESP32 -----
GPIO.enable_wlts = (1 << 2);   // pin ca iesire
GPIO.out_wlts    = (1 << 2);   // pin pe HIGH (write-1-to-set, ATOMIC)
GPIO.out_wltc    = (1 << 2);   // pin pe LOW (write-1-to-clear)

// ----- PIC (seria 16F) -----
TRISB &= ~(1 << 5);           // pin ca iesire (0 = iesire la PIC!)
LATB  |= (1 << 5);             // pin pe HIGH
LATB  &= ~(1 << 5);           // pin pe LOW
```

Aspect	AVR	STM32	ESP32	PIC
Lățime registru	8 biți	32 biți	32 biți	8 biți
Direcție pin	DDRx , 1 = ieșire	MODER / CRL	enable	TRISx , 0 = ieșire
Scriere nivel	PORTx	ODR sau BSRR	out	LATx
Citire pin	PINx	IDR	in	PORTx
Setare atomică	sbi (automat)	BSRR	wlts / wltc	-
Ceas periferic	mereu pornit	trebuie activat (RCC)	gestionat de SDK	mereu pornit

OBSERVAȚIA CEA MAI IMPORTANTĂ

Controlerele moderne oferă regiștri

DEDICAȚI

pentru setare și ștergere (BSRR la STM32, wlts/wltc la ESP32). Scrierea în ei este o singură operație, deci **ATOMICĂ PRIN CONSTRUCȚIE**

- problema din secțiunea 11 dispăre complet. Existența acestor regiștri este dovada că problema RMW este reală și suficient de importantă încât producătorii să adauge hardware special pentru ea.

CAPCANĂ LA PORTARE

La PIC, TRISx folosește convenția

INVERSĂ

față de AVR: 0 înseamnă ieșire, 1 înseamnă intrare. Codul portat mecanic funcționează exact pe dos. Verificați întotdeauna convenția în datasheet, nu presupuneți.

15 Cod sursă

Afișarea binară a unei valori

afisare_binara.ino

```
// Legati 6 LED-uri pe pinii 8...13, adica pe bitii 0...5 ai portului B.
// Bara de LED-uri devine "afisajul binar" al oricarei valori pe 6 biti.

void arata(uint8_t v) {
    PORTB = v & 0x3F;    // bitii 0-5 ai valorii ajung pe cele 6 LED-uri
    _delay_ms(1500);    // pauza, ca sa aveti timp sa cititi bara
}

int main(void) {
    DDRB = 0x3F;        // toti cei 6 pini, iesiri

    uint8_t reg = 0b00001010;
    arata(reg);          // initial : 001010

    reg |= (1 << 5);    arata(reg); // set b5 : 101010
    reg &= ~(1 << 1);    arata(reg); // clr b1 : 101000
    reg ^= (1 << 3);    arata(reg); // tgl b3 : 100000

    // TESTAREA unui bit: LED-ul de pe pinul 7 arata raspunsul
    DDRD |= (1 << PD7);
    if (reg & (1 << 5)) PORTD |= (1 << PD7); // bitul 5 este SETAT
    else PORTD &= ~(1 << PD7); // bitul 5 este sters

    while (1) { }
}
```

Exerciții de manipulare

exercitii_biti.ino

```
// 1. Numararea bitilor setati (algoritmul lui Kernighan)
uint8_t numaraBiti(uint8_t v) {
    uint8_t n = 0;
    while (v) {
        v &= (v - 1);    // sterge cel mai din dreapta bit setat
        n++;
    }
    return n;
}

// 2. Testarea daca o valoare este putere a lui 2
bool estePutereDe2(uint8_t v) {
    return v && !(v & (v - 1));
}

// 3. Inversarea ordinii bitilor
uint8_t inverseazaBiti(uint8_t v) {
    uint8_t r = 0;
    for (uint8_t i = 0; i < 8; i++) {
        r = (r << 1) | (v & 1);
        v >>= 1;
    }
}
```

```

    return r;
}

// 4. Schimbarea a doua valori fara variabila temporara
void interschimba(uint8_t *a, uint8_t *b) {
    *a ^= *b;  *b ^= *a;  *a ^= *b;
}

// 5. Extragerea nibble-ului superior si inferior
uint8_t nibbleSus(uint8_t v) { return (v >> 4) & 0x0F; }
uint8_t nibbleJos(uint8_t v) { return  v      & 0x0F; }

```

Aceeași aplicație, trei niveluri de abstractizare

trei_niveluri.ino

```

// ===== NIVELUL 1: biblioteca Arduino =====
// avantaj: lizibil imediat   dezavantaj: lent, neportabil
void blinkArduino() {
    pinMode(13, OUTPUT);
    digitalWrite(13, HIGH);
    delay(500);
    digitalWrite(13, LOW);
    delay(500);
}

// ===== NIVELUL 2: registre AVR, cu nume =====
// avantaj: rapid, explicit   dezavantaj: legat de AVR
void blinkRegistri() {
    DDRB |= (1 << PB5);
    PORTB |= (1 << PB5);
    _delay_ms(500);
    PORTB &= ~(1 << PB5);
    _delay_ms(500);
}

// ===== NIVELUL 3: strat de abstractizare propriu =====
// avantaj: rapid SI portabil - se schimba doar acest bloc la portare
#if defined(__AVR__)
    #define LED_DIR_REG    DDRB
    #define LED_OUT_REG    PORTB
    #define LED_PIN_NR    PB5
#elif defined(ESP32)
    #define LED_DIR_REG    GPIO.enable
    #define LED_OUT_REG    GPIO.out
    #define LED_PIN_NR    2
#endif

#define LED_INIT()    (LED_DIR_REG |= (1UL << LED_PIN_NR))
#define LED_ON()      (LED_OUT_REG |= (1UL << LED_PIN_NR))
#define LED_OFF()     (LED_OUT_REG &= ~(1UL << LED_PIN_NR))
#define LED_TOGGLE()  (LED_OUT_REG ^= (1UL << LED_PIN_NR))

void blinkPortabil() {
    LED_INIT();
    LED_ON();    _delay_ms(500);
    LED_OFF();   _delay_ms(500);
}

```

NIVELUL 3 ESTE ȚINTA

Acesta este tiparul folosit în industrie: logica aplicației apelează `LED_ON()`, iar portarea pe alt microcontroler înseamnă rescrierea unui singur bloc de definiții. Veți folosi această structură la proiectul final.

Măsurarea diferenței de performanță

benchmark_bit1.ino

```
// Nu avem inca monitorul serial (lucrarea 6), deci nu tiparim durata:
// o facem VIZIBILA. Fiecare metoda tine aprins un LED-martor cat timp
// ruleaza bucla. Cu cat metoda e mai rapida, cu atat martorul clipeste
// mai scurt. Pe osciloscop, latimea impulsului ESTE durata masurata.

#define N 20000

int main(void) {
    DDRB |= (1 << PB0); // pinul care se comuta
    DDRD |= (1 << PD5) | (1 << PD6) | (1 << PD7); // cei trei martori

    uint16_t i;
    while (1) {
        PORTD |= (1 << PD7); // varianta cu functie
        for (i = 0; i < N; i++) { digitalWrite(8, HIGH); digitalWrite(8, LOW); }
        PORTD &= ~(1 << PD7);
        _delay_ms(300);

        PORTD |= (1 << PD6); // varianta cu registru
        for (i = 0; i < N; i++) { PORTB |= (1 << PB0); PORTB &= ~(1 << PB0); }
        PORTD &= ~(1 << PD6);
        _delay_ms(300);

        PORTD |= (1 << PD5); // comutare prin scriere in PINB
        for (i = 0; i < N; i++) { PINB = (1 << PB0); PINB = (1 << PB0); }
        PORTD &= ~(1 << PD5);
        _delay_ms(300);
    }
}
```

TRUCUL CU PINB

Pe AVR, scrierea unui 1 în registrul PINx

COMUTĂ

bitul corespunzător din PORTx. Este cea mai rapidă metodă de comutare a unui pin - o singură instrucțiune, atomică. Detaliu care se găsește doar în datasheet, niciodată într-un tutorial de Arduino.

16 Montaj interactiv

Aici vedeți simultan cele trei fețe ale aceluiași lucru: instrucțiunea C pe care o scrieți, bitul care se schimbă în registru și LED-ul care se aprinde. Panoul de regiștri este și interactiv - dați clic direct pe un bit ca să-l comutați manual și priviți ce se întâmplă cu montajul.

► CELE PATRU IDIOMURI FUNDAMENTALE, VĂZUTE PE O BARĂ DE LED-URI

idiomuri.ino

```
1  #include <avr/io.h>
2  #include <util/delay.h>
3
4  /* Pinii 8...13 sunt bitii 0...5 ai portului B.
5     Pinul 7 (PD7) este un LED-martor pentru idiomul de TESTARE. */
6
7  int main(void)
8  {
9      DDRB = 0x3F;           // toti cei 6 pini devin iesiri
10     DDRD |= (1 << PD7);    // pinul martor, iesire
11
12     while (1) {
13         PORTB = 0x00;      // pornim de la 0b000000
14         _delay_ms(600);
15
16         PORTB |= (1 << PB0); // SETARE bit 0 -> aprinde LED 0
17         _delay_ms(600);
18
19         PORTB |= (1 << PB2) | (1 << PB4); // doua SETARI odata
20         _delay_ms(600);
21
22         PORTB &= ~(1 << PB2); // STERGERE bit 2 -> stinge LED 2
23         _delay_ms(600);
24
25         PORTB ^= (1 << PB4); // COMUTARE bit 4 -> se inverseaza
26         _delay_ms(600);
27
28         if (PORTB & (1 << PB0)) // TESTARE bit 0
29             PORTD |= (1 << PD7); // adevarat -> martorul se aprinde
30         else
31             PORTD &= ~(1 << PD7);
32         _delay_ms(900);
33     }
34 }
```

DIGITAL
13 12 11 10 9 8 7 6 5 4 3 2 1 0
ANALOGIC
A0 A1 A2 A3 A4 A5
0.000 s

Bară de 6 LED-uri pe portul B

pinii D8, D9, D10, D11, D12, D13

D8

D9

D10

D11

D12

D13

0b000000 = 0 în zecimal

Martor: bitul 0 este 1

pinul D7

stins

0.000 s timp simulat 0 întreruperi tratate în pauză

REGIȘTRII IMPLICAȚII - DAȚI CLIC PE UN BIT CA SĂ-L COMUTAȚI

DDRB = 0x00 = 0 = 0b00000000

7 0 DDB7	6 0 DDB6	5 0 DDB5 D13	4 0 DDB4 D12	3 0 DDB3 D11	2 0 DDB2 D10	1 0 DDB1 D9	0 0 DDB0 D8
----------------	----------------	-----------------------	-----------------------	-----------------------	-----------------------	----------------------	----------------------

PORTB = 0x00 = 0 = 0b00000000

7 0 PORTB7	6 0 PORTB6	5 0 PORTB5 D13	4 0 PORTB4 D12	3 0 PORTB3 D11	2 0 PORTB2 D10	1 0 PORTB1 D9	0 0 PORTB0 D8
------------------	------------------	-------------------------	-------------------------	-------------------------	-------------------------	------------------------	------------------------

Versiune interactivă pe site.

DE ÎNCERCAT

Sub bară vi se arată tot timpul valoarea binară și zecimală a portului: urmăriți cum fiecare idiom schimbă exact un bit. Opriți execuția și dați clic pe biții din

PORTB

: LED-urile se aprind și se sting imediat. Apoi dați clic pe biții din

DDRB

- dacă un bit din DDRB este 0, pinul devine intrare și LED-ul nu se mai aprinde, oricât ați insista pe PORTB. Aceasta este toată diferența dintre cele două registre.

► EXERCIȚIU - LUMINA CARE BALEIAZĂ („KNIGHT RIDER”)

baleiaj.ino

```

1  #include <avr/io.h>
2  #include <util/delay.h>
3
4  int main(void)
5  {
6      DDRB = 0x3F;
```

```
7   unsigned char pozitie = 0;
8
9   while (1) {
10      /* Aprindeti UN SINGUR LED, cel de pe pozitia curenta.
11         Indiciu: PORTB = (1 << pozitie);
12         Apoi mutati pozitia mai departe si, cand ajunge la 5,
13         luati-o de la capat. Pas de 120 ms.
14
15         Provocare: faceti-l sa se intoarca, nu sa sara inapoi. */
16
17      _delay_ms(120);
18   }
19 }
```

DIGITAL 13 12 11 10 9 8 7 6 5 4 3 2 1 0

ANALOGIC A0 A1 A2 A3 A4 A5

0.000 s

Bară de 6 LED-uri pe portul B

pinii D8, D9, D10, D11, D12, D13



0b000000 = 0 în zecimal

0.000 s timp simulat 0 întreruperi tratate în pauză

REGIȘTRII IMPLICAȚII - DAȚI CLIC PE UN BIT CA SĂ-L COMUȚĂȚI

PORTB = 0x00 = 0 = 0b00000000

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
PORTB7	PORTB6	PORTB5 D13	PORTB4 D12	PORTB3 D11	PORTB2 D10	PORTB1 D9	PORTB0 D8

Versiune interactivă pe site.

DE ÎNCERCAT

Un singur LED aprins înseamnă un singur bit pe 1 în PORTB. Deplasarea la stânga cu << mută acel bit - de aceea efectul se obține dintr-o singură linie.

17 Sarcini de lucru

☐ Converteți manual, fără calculator: 0b11010110 în hexa și zecimal; 0x3F în binar.

☐ Implementați funcția de afișare binară și verificați-o pentru 0, 1, 128 și 255.

- ☐ Scrieți cele patru macrouri (SET, CLR, TGL, CHK) și folosiți-le pentru a comanda un LED.
- ☐ Setati simultan biții 1, 3 și 5 dintr-o variabilă, printr-o singură expresie.
- ☐ Ștergeți un câmp de 3 biți și scrieți în el valoarea 5, fără a afecta restul registrului.
- ☐ Extrageți nibble-ul superior și pe cel inferior dintr-un octet și afișați-le separat.
- ☐ Implementați numărarea biților setați și verificați rezultatul pentru 0b10110110.
- ☐ Rulați benchmark-ul și notați raportul dintre cele trei metode de comutare a unui pin.
- ☐ Scrieți același program de blink în cele trei niveluri de abstractizare și comparați dimensiunea codului compilat.
- ☐ Identificați în datasheet-ul ATmega328P adresa registrului PORTD și accesați-l prin pointer, fără a folosi numele.

18 Provocare

BIBLIOTECĂ PROPRIE DE ACCES LA PINI

Scrieți un mic strat de abstractizare care oferă funcțiile `pinInit(port, bit, dir)`, `pinWrite(port, bit, val)`, `pinRead(port, bit)` și `pinToggle(port, bit)`, folosind exclusiv operații pe biți și pointeri către regiștri.

Cerințe: să funcționeze pe toate cele trei porturi (B, C, D), să fie de cel puțin 10 ori mai rapidă decât `digitalWrite()` și să nu folosească nicio funcție din biblioteca Arduino. Comparați dimensiunea codului compilat cu varianta care folosește biblioteca standard.

Extindere: adăugați o variantă `pinToggleAtomic()` care folosește trucul cu `PINx` și demonstrați, cu un analizor logic sau cu un al doilea pin, că este atomică.

19 Întrebări de verificare

Ce produce expresia $\sim(1 \ll 5)$ pe 8 biți?

0b00100000

0b11011111

0b00000101

0b11111111

Care idiom șterge un bit fără a-i afecta pe ceilalți?

$\text{REG} |= (1 \ll n)$

$\text{REG} \&= \sim(1 \ll n)$

$\text{REG} ^= (1 \ll n)$

$\text{REG} = (1 \ll n)$

Câte instrucțiuni mașină generează de regulă $\text{REG} |= \text{mask}$ pentru un registru din memoria extinsă?

Una singură

Două

Trei: citește, modifică, scrie

Depinde de valoarea măștii

De ce sunt regiștrii declarați volatili în fișierele antet?

Ca să ocupe mai puțină memorie

Pentru că valoarea lor se poate schimba independent de program, prin hardware

Pentru viteză

Nu sunt declarați volatili

Ce face scrierea unui 1 în registrul PINx pe AVR?**De ce controlerele STM32 au registrul BSRR separat de ODR?****Cum se citește corect un câmp aflat pe pozițiile 6–7 ale unui registru?****La PIC, ce valoare din TRISx configurează un pin ca ieșire?**

Răspunsuri: 1-b, 2-b, 3-c, 4-b, 5-b, 6-b, 7-b, 8-b

1. $1 \ll 5$ dă 0b00100000; negarea inversează toți biții, rezultând 0b11011111 - masca folosită pentru ștergerea bitului 5.
2. Masca $\sim(1 \ll n)$ are toți biții 1 cu excepția bitului n. ȘI cu 1 nu schimbă nimic, iar ȘI cu 0 forțează pe 0 exact bitul dorit.
3. Este o operație citește–modifică–scrie: încărcare din memorie, aplicarea măștii, scriere înapoi. Fereastra dintre pasul 2 și 3 permite coruperea de către o întrerupere.
4. PINx reflectă starea fizică a pinilor, iar fanioanele se setează de hardware. Fără volatile, compilatorul ar refolosi o valoare citită anterior, iar buclele de așteptare nu s-ar termina niciodată.
5. Este o facilitate specifică AVR: scrierea unui 1 în PINx comută bitul din PORTx, într-o singură instrucțiune atomică - cea mai rapidă metodă de toggle.
6. BSRR permite setarea sau ștergerea unor biți printr-o singură scriere, fără citire prealabilă. Elimină complet problema citește–modifică–scrie.
7. Se deplasează întâi câmpul până la poziția 0, apoi se maschează cu 0x03 pentru a izola cei doi biți.
8. Convenția PIC este inversă față de AVR: 0 înseamnă ieșire (Output), 1 înseamnă intrare (Input). Cod portat mecanic din AVR funcționează exact pe dos.

20 Resurse

Cursul 2 - arhitectura microcontrolerului

Îndrumarul complet de laborator

Cursul 3 - programarea microcontrolerului

Pagina cursului - toate materialele