Лекція 8

1. Використання стеку.

Стек розміщується в SRAM. Він представляє собою LIFO-буфер (last input first output, останнім увійшов – першим вийшов). Це як колода карт, зверху кладете і зверху ж берете. Початок стека розміщується унизу SRAM і по мірі його заповнення він росте в напрямку початку SRAM. Керування стеком виконується за допомогою спеціального 2 - байтного регістра SP, що розміщується у 2-х регістрах SPH (старший байт) та SPL (молодший байт). Деякі молодші моделі мають в наявності лише SPL. Значення стеку за замовчуванням рівне 0x0000, тому стек необхідно проініціалізувати вручну. Звісно вибір початку розміщення вершини стеку може бути довільним, однак, якщо його розмістити доволі близько до оперативних даних, тобто ближче до початку SRAM, тоді, по мірі збільшення стеку, може бути його «наїзд» на дані у SRAM, що призведе до їх спотворення та до не правильної роботи програми. Тому стек розміщують у кінці SRAM.

```
ldi     r16, Low(RAMEND)
out     SPL, r16
ldi     r16, High(RAMEND)
out     SPH, r16
```

Значення константи RAMEND залежить від моделі МК та зберігається у конфігураційному файлі цієї моделі.

Стек неявно використовується підпрограмами через команди call, rcall, icall, ret, reti та перериваннями для збереження зворотної адреси точки виклику в програмі. Також стек можна використовувати для швидкого зберігання байтів інформації. Наприклад, нам необхідно регістр для виконання певних операцій, а вільного немає. Тоді вибираємо один з регістрів, що на даний момент не використовується, зберігаємо його значення у стек, працюємо з ним, а по закінченню маніпуляцій з ним відновлюємо зі стеку його попереднє значення.

Для роботи зі стеком призначені дві команди:

push – заносить значення регістра загального призначення у стек (push register on stack), 2 такти;

pop – витягує значення зі стеку у регістр загального призначення (pop register from stack), 2 такти;

Ці команди працюють лише через регістри загального призначення, і якщо необхідно зберегти у стеку значення регістра вводу/виводу, то його перед тим необхідно скопіювати у регістр загального призначення.

При використанні стеку необхідно чітко контролювати логіку роботи з ним, та особливо, щоб не трапився неконтрольований збій у його роботі. Найпростіший випадок, це якщо ми запхали дані в одному порядку, а витягнули не в тому , якому б хотіли. Неконтрольований збій – це якщо ми запхали дані у стек і забули витягнути, або навпаки, витягнули більше, аніж запхали. Тоді відбувається зміщення лічильника стеку, і ми можемо, наприклад, своїми діями переписати збережені адреси повернення підпрограми, чи, якщо ця помилка трапляється циклічно, запороти усі дані в SRAM.

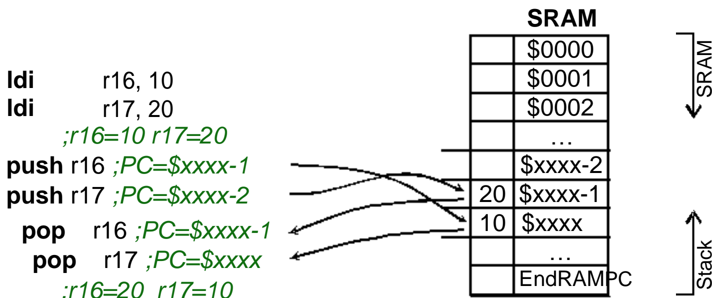


Рис. 1. Обмін двох регістрів значеннями через стек

На рис. 1 наведено приклад явного використання стеку для обміну регістрами між собою значеннями без використання третього проміжного регістра.

2. Підпрограми

Підпрограми це окремо виділені куски програмного коду, які часто використовуються. Виклик підпрограми здійснюється за допомогою однієї з команд:

call – виконує виклик підпрограми в межах цілої програмної пам'яті. Команда займає у програмі 4 байти (32 біти), 22 біти з яких відведені під адресу, однак, для переважної більшості моделей програмний лічильник займає 16 біт, тому ця команда може адресувати для цих моделей лише до 128 кБайт FLASH. Команда виконується за 4 такти.

rcall – виконує відносний виклик підпрограми, здійснюючи зміщення вперед чи назад у програмній пам'яті на вказане число відносно значення поточної адреси (програмного лічильника). Значення зміщення адреси програмного лічильника може приймати -2047… +2048 у пам'яті програм. Команда виконується за 3 такти.

icall – виконує непрямий виклик підпрограми, значення адреси якої зберігається в індексному двобайтному регістрі Z. Об’єм FLASH для цієї команди обмежується 128 кБайтами. Виконується за 3 такти.

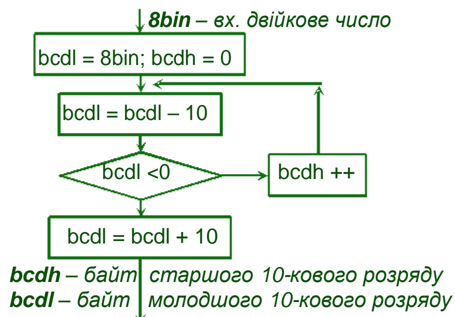
При виклику підпрограми за допомогою однієї з команд у стек автоматично заноситься адреса (2 байти) наступної команди після команди виклику. Виконання підпрограми завершується командою **ret** (4 такти), яка присвоює програмному лічильнику значення зворотної адреси, що витягується зі стеку.

Для команд виклику адреси підпрограм вказують за допомогою міток, а компілятор вже сам обчислює потрібні значення.

При використанні підпрограм необхідно обов’язково проініціалізувати стек (!).

```
.include "m32Adef.inc"
.def    _8bin       =r18
.def    _bcdlow     =r18
.def    _bcdhigh    =r19

        .CSEG
        ;Ініціалізація стеку
        ldi     r16, Low(RAMEND)
        out     SPL, r16
        ldi     r16, High(RAMEND)
        out     SPH, r16
        …
        .org    $100
 $100:  ldi     _8bin, 35       ;r18 = 35
 $101:  rcall   BCD             ;$102 → STACK;   PC=$150
 $102:  nop                     ;r19 = 3; r18 = 5
 $103:  ldi     _8bin, 71       ;r18 = 71
 $104:  rcall   BCD             ;$105 → STACK;   PC=$150
 $105:  nop                     ;r19 = 7; r18 = 1
        …
;Підпрограма Двійково-Десяткового Кодування (до 99, незапаковане)
        .org    $150
BCD:    clr     _bcdhigh
BCD1:   subi    _bcdlow, 10
        brcs    BCD2
        inc     _bcdhigh
        rjmp    BCD1
BCD2:   subi    _bcdlow, -10
        ret     ; PC ← STACK
```



3. Реалізація переривань

У МК AVR переривання реалізують механізм оброблення подій від вбудованих периферійний пристроїв. Оскільки моделі AVR відрізняються одна від одної кількістю та різноманітністю вбудованих пристроїв, то відповідно, і кількість переривань у них є різною.

Основною відправною точкою механізму переривань є таблиця векторів переривань. Ця таблиця представляє собою послідовний список адрес у програмній пам'яті (рис. 3.12). Як правило, ця таблиця розміщується на початку адресного простору пам'яті програм. Інформацію про наявні переривання та адреси їхніх векторів переривань можемо отримати в інструкції виробника на конкретну модель.

Address	Labels	Code	Comments
$000	jmp	RESET	; Reset Handler
$002	jmp	EXT_INT0	; IRQ0 Handler
$004	jmp	EXT_INT1	; IRQ1 Handler
$006	jmp	EXT_INT2	; IRQ2 Handler
$008	jmp	TIM2_COMP	; Timer2 Compare Handler
$00A	jmp	TIM2_OVF	; Timer2 Overflow Handler
$00C	jmp	TIM1_CAPT	; Timer1 Capture Handler
$00E	jmp	TIM1_COMPA	; Timer1 CompareA Handler
$010	jmp	TIM1_COMPB	; Timer1 CompareB Handler
$012	jmp	TIM1_OVF	; Timer1 Overflow Handler
$014	jmp	TIM0_COMP	; Timer0 Compare Handler
$016	jmp	TIM0_OVF	; Timer0 Overflow Handler
$018	jmp	SPI_STC	; SPI Transfer Complete Handler
$01A	jmp	USART_RXC	; USART RX Complete Handler
$01C	jmp	USART_UDRE	; UDR Empty Handler
$01E	jmp	USART_TXC	; USART TX Complete Handler
$020	jmp	ADC	; ADC Conversion Complete Handler
$022	jmp	EE_RDY	; EEPROM Ready Handler
$024	jmp	ANA_COMP	; Analog Comparator Handler
$026	jmp	TWI	; Two-wire Serial Interface Handler
$028	jmp	SPM_RDY	; Store Program Memory Ready Handler

Рис. 2. Таблиця векторів переривань для ATmega32A

Для того, щоб певна подія для вбудованого пристрою МК могла бути згенерованою, необхідно перш за все активізувати цей периферійний пристрій та дати дозвіл на переривання для цієї події, а також дати загальний дозвіл на переривання у регістрі стану SREG.

При виникненні цієї події виставляється прапорець переривання у відповідному регістрі, і при першій ж можливості, після виконання поточної команди, і якщо немає у цей час іншого переривання, заноситься у стек адреса наступної команди та виконується перехід на адресу вектора переривань для події, при цьому апаратно скидається

прапор переривання. Далі за цієї адресою вектора у таблиці переривань розміщується команда переходу jmp чи rjmp, яка переходить на підпрограму переривання . Виконання підпрограми завершується командою reti, витягується зі стеку адреса повернення та виконується перехід в основну програму. Ці підпрограми, як правило, розміщують одразу ж після таблиці переривань.

У випадку, якщо подія трапилася, виставився для неї відповідний прапор переривання, але загальний дозвіл ще не наданий (в регістрі стану SREG), тоді ця подія буде оброблена при наданні відповідного загального дозволу переривання.

Слід зазначити, що для деяких подій немає відповідних прапорців переривань. Для них переривання генеруються протягом усього часу, допоки присутня відповідна умова, що необхідна для генерації переривання. Відповідно, якщо умови, що викликають переривання, щезнуть до надання дозволів на переривання, то генерації переривання не відбудеться.

Якщо переривання генерується безперестанно, тобто маємо постійну умову для безпрапорцевих переривань чи в черзі стоять переривання з виставленими прапорцями, то між генеруваннями переривань виконується одна команда з основного коду.

Покажемо роботу механізму переривань на прикладі «**зовнішніх переривань**».

Зовнішні переривання це реакція МК на зміну рівня сигналу на його виводах. Такі переривання бувають індивідуальними (INT), тобто для конкретного одного виводу МК, та груповими (PCINT), тобто одне переривання для групи виводів. МК серії tiny мають як мінімум хоча б одне INT0, МК серії mega мають як мінімум два INT0 та INT1. Групові переривання (PCINT) вже залежать від конкретної моделі, і тому можуть бути, а можуть і ні.

Індивідуальні зовнішні переривання (INT) можуть генеруватися при наявності зростаючого чи спадаючого фронту сигналу, або постійного низького рівня на виводі МК (рис. 3). При налаштуванню на низький рівень прапори переривання не виставляються та переривання генеруються допоки присутня умова низького рівня.

Групове зовнішнє переривання (PCINT) охоплює, як правило, виводи цілого порту. При цьому ми можемо індивідуально вибирати, які виводи порту будуть генерувати переривання, а які ні . Генерування переривання буде відбуватися при будь-якій зміні рівня сигналу (при зростанні та спаданні фронтів) на будь-якому виводі групи PCINT.

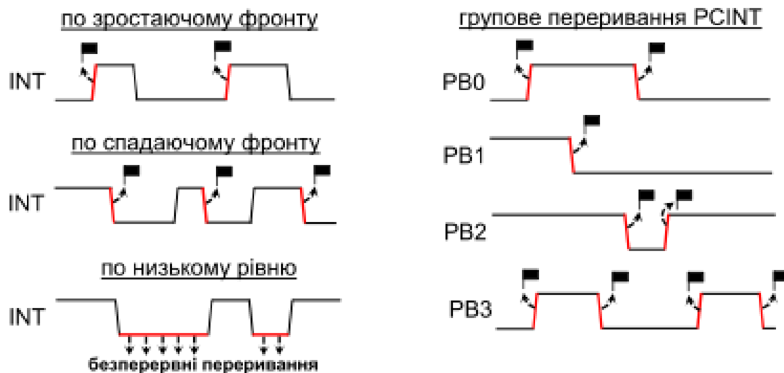


Рис. 3. Діаграми генерування зовнішніх переривань

Модель ATmega32A має в наявності 3 зовнішні переривання INT0, INT1 та INT2. INT0 та INT1 можуть бути налаштовані на переривання будь-якої з трьох умов, INT2 – лише на реагування зростаючого чи спадаючого фронту сигналу. Наприклад, у моделі ATmega324P переривання INT2 може бути налаштоване на усі три умови.

Для роботи зі зовнішніми перериваннями в ATmega32A використовуються 4 регістри:

GICR (даються дозволи на переривання INT0 , INT1 та INT2);

GIFR (виставляються прапорці переривань);

MCUCR (налаштовуються умови для переривань INT0 та INT1);

MCUCSR (налаштовується умова для переривання INT2).

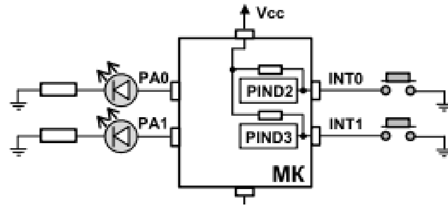
Назви цих регістрів у різних моделей МК можуть бути різними, найчастіше співпадають назви бітів для таких регістрів.

Таблиця 1. Умови генерації зовнішніх переривань для ATmega32A

	INT0		**INT1**		**INT2**
	ISC01	ISC00	ISC11	ISC10	ISC2
по низькому рівню	0	0	0	0	-
по спадаючому фронту	1	0	1	0	0
по зростаючому фронту	1	1	1	1	1

Продемонструємо роботу зовнішніх переривань на прикладі. Підключимо до виводів INT0 та INT1 кнопки, а самі виводи налаштуємо на вхід з підключеними внутрішніми підтягуючими резисторами. INT0 налаштуємо на переривання по спаду фронту сигналу на виводі, а INT1 на переривання по зростанню фронту сигналу. Переривання

повинні інвертувати сигнали на виходах, відповідно, PA0 та PA1 порту A (світлодіоди будуть засвічуватися/гаснути).




```
                .CSEG
                .org    $000
                jmp     RESET       ;Reset Handler
                .org    $002
                jmp     EXT_INT0    ;IRQ0 Handler
                .org    $004
                jmp     EXT_INT1    ;IRQ1 Handler

                .org    $028
                reti                ;Store Program Memory Ready Handler
;-------- Підпрограми переривань --------
EXT_INT0:       sbic    PORTA, 0
                rjmp    elsePA0
                sbi     PORTA, 0
                reti
elsePA0:        cbi     PORTA, 0
                reti

EXT_INT1:       sbic    PORTA, 1
                rjmp    elsePA1
                sbi     PORTA, 1
                reti
elsePA1:        cbi     PORTA, 1
                reti
;--------------------------------------------------

                ;ініціалізація стека
RESET:          ldi     r16, Low(RAMEND)
                out     SPL, r16
                ldi     r16, High(RAMEND)
                out     SPH, r16

                ldi     r16, 0x00
                ldi     r17, 0xFF
```

```
        out     DDRA, r17           ;Порт A на вихід
        out     PORTA, r16
        out     DDRD, r16           ;Порт D на вхід
        out     PORTD, r17

;Зовнішні переривання INT0, INT1
        ldi     r16, (1<<INT0)|(1<<INT1)      ;вибір переривань
        out     GICR, r16

        ldi     r16, (1<<ISC01) | (0<<ISC00) | (1<<ISC11) | (1<<ISC10)
        out MCUCR, r16 ;INT0-по зрост.рівню; INT1-по спад.фронту
;скид прапорців зовнішнього переривання
        ldi     r16, (1<<INTF0) | (1<<INTF1)
        out     GIFR, r16

;загальний дозвіл на переривання
        sei
```

```
main:
        rjmp    main
```

У своїй програмі ми не використовуємо усіх наявних для вказаної моделі МК переривань, і тому на початку коду у таблиці переривань ми вписуємо лише вектори рестарту МК та тих переривань, які ми будемо використовувати. Однак правилом хорошого тону є вписувати також останній у таблиці вектор переривання, свого роду як заглушку, що позначає кінець таблиці. Для останнього вектора немає підпрограми, а лише команда виходу з переривання. Це гарантує, що ми не будемо вписувати на місці таблиці переривань свій програмний код.

У таблиці переривань ми можемо використовувати не абсолютну команду переходу jmp, а відносну rjmp, якщо впевнені, що підпрограми переривань знаходяться на віддалі 2 кбайт. Це нам зекономить по 1 такту.

При використанні переривань, обов’язково необхідно проініціалізувати стек, оскільки він використовується для збереження зворотної адреси.

Ще один момент стосується регістру стану SREG. Якщо програма переривання використовує команди, що виставляють прапорці в регістрі стану, тоді перед початком роботи в перериванні необхідно зберегти SREG у стек, а при виході із переривання, відновити його значення зі стеку. Це необхідно зробити, оскільки переривання вклинюються в основну програму, і може трапитися так, що ми командою

порівняння чисел cp виставили прапорці, і хочемо на їхній основі здійснити перехід за допомогою команд Branch, але тут програма пішла на переривання, в якому її підпрограма змінила прапорці у регістрі стану SREG, і при поверненні в основну програму буде вже спотворена логіка для умовного переходу. Тому при перериваннях необхідно зберігати SREG у стеку.

```
Interrupt_subroutine:
        in      r16, SREG       ;рег.заг.призн. ← SREG
        push    r16             ;рег.заг.призн. → стек
        …  програмний код переривання
        …
        pop     r16             ;рег.заг.призн. ← стек
        out     SREG, r16       ;рег.заг.призн. → SREG
        reti                    ;вихід з переривання
```

4. Макроси

Використання асемблерних макросів з першого погляду може здатися схожим на використання підпрограм, що викликаються за допомогою команд call та rcall. Однак, на відміну від підпрограм, вони:

– Не мають окремого адресного розташування за межами основної програми, куди ми переходимо командами переходу. Програмний код макросу підставляється на етапі компіляції у точку виклику. Таким чином, з метою читабельності коду, ми виносимо часто повторювані куски коду в макроси, а під час компіляції вони назад підставляються.

Питання: чому ж тоді не оформити ці куски коду в підпрограми? Переходи на підпрограми займають час, а якщо у нас постійно повторюється 3-4 команди, то тоді немає змісту оформляти це у вигляді підпрограми, і тому легше винести у вигляді макросу. По друге, макроси можуть давати нашій програмі певну гнучкість (про це далі).

Звісно, у порівнянні з підпрограмами, виклики макросів збільшують об'єм програмної пам'яті, тобто, скільки раз викликали макрос, на стільки ж (к-сть викликів * об'єм макросу) і збільшується використання програмної пам'яті. У той час, як підпрограми при багатократному виклику не потребують додаткової пам'яті.

– Макроси, на відміну від підпрограм, можуть зберігатися в окремих файлах. Тобто, ми можемо з легкістю використовувати готові напрацювання, оформлені у вигляді макросів, в інших проектах, чи використовувати у своїх проектах готові «чужі» макроси.

```
.macro  R16mult10                    ;r20 = r20*10 = (r20<<1)+(r20<<3)
        lsl     r20
        mov     _temp, r20
        lsl     r20
        lsl     r20
        add     r20, _temp
.endmacro

.def    _temp = r16

        .CSEG
        …
        ldi     r20, 14              ;r20 = 14
        R16mult10                    ;r20 = r20*10 = 140
        subi    r20, 121             ;r20 = r20 – 121 = 19
        R16mult10                    ;r20 = r20*10 = 190
        subi    r20, 180             ;r20 = r20 – 180 = 10
        R16mult10                    ;r20 = r20*10 = 100
        …
```

У наведеному лістингу програми ми проілюстрували використання макросу на прикладі швидкого множення беззнакового однобайтного числа на 10.

– Макроси можуть мати вхідні параметри, що робить їх більш універсальними, аніж використання підпрограм через команди переходу. Як параметри, ми можемо передати у макрос або константи, або назви регістрів. Макрос може приймати до 10 параметрів. Посилання на ці параметри позначаються в середині макросу як @0-@9. Порядок слідування визначається при виклику макросу. Під час компіляції при підстановці коду макросу у точки виклику замість параметризованих змінних підставляються параметри, що вказані через кому після імені викликаного макросу.

```
.include "m32Adef.inc"
.def    _temp = r16

.macro  outi
        ldi     _temp, @1
    .if @0 < 0x40
        out     @0, _temp
    .else
        sts     @0, _temp
    .endif
.endm
```

```
.macro  addi
        ldi     _temp, @1
        add     @0, _temp
.endm

.macro  ldi2
        ldi     _temp, @1
        mov     @0, _temp
.endm

        .CSEG
        outi    DDRA, 0xFF
        outi    PORTA, 0x00

        outi    DDRB, 0x00
        outi    PORTB, 0b01010101

        ldi2    r0, 50          ;r0 = 50
        addi    r0, 25          ;r0 = r0 + 25 = 75

        ldi2    r1, 50          ;r1 = 50
        addi    r1, -25         ;r1 = r1 + -25 = 25
```

Регістр								
DDRA	1	1	1	1	1	1	1	1
PORTA	0	0	0	0	0	0	0	0
DDRB	0	0	0	0	0	0	0	0
PORTB	0	1	0	1	0	1	0	1

У цьому прикладі ми створили 3 макроси, що мають полегшувати процес програмування та робити читабельнішим наш асемблерний код.

Перший макрос outi виводить у будь-який регістр вводу/виводу константу. Зауважте, ми використали у ньому умовну компіляцію. Як ми знаємо, у нафаршированих периферією МК частина регістрів вводу/виводу знаходяться у пам’яті даних після адреси 0x005F, для яких не працюють команди out та in. При роботі з такими регістрами необхідно напряму звертатися до SRAM через відповідні команди доступу до пам’яті. Тому наш макрос спрощує нам задачу запису константи у регістр вводу/виводу, і нам не потрібно слідкувати за тим, в якій області пам’яті знаходиться цей регістр.

Другий макрос addi реалізує команду додавання до будь-якого регістра загального призначення константи.

Третій макрос ldi2 дає можливість присвоювати константу як старшій половині файлу регістрів загального призначення, так і його молодшій частині.

5. Робота з даними у SRAM, FLASH та EEPROM.

SRAM. Змінні величини у програмі мовою асемблер ми закріплюємо, як правило, за вільними регістрами загального призначення. Якщо кількість змінних є більшою за кількість регістрів, то ми можемо, наприклад, один регістр використовувати почергово між декількома змінними, і в той час, коли одна змінна використовується з регістром, решта змінних зберігаються у стеку . Цей спосіб потребує ретельного контролю за переміщенням даних у стеку. Тому простіше, при нестачі вільних регістрів загального призначення, розміщувати наші змінні безпосередньо в оперативній пам'яті (SRAM). Звісно, для роботи з нашими змінними в SRAM, необхідно їх буде переміщувати в регістри загального призначення, модифікувати, і назад зберігати в пам'ять даних. Для цього слід передбачити декілька регістрів, що будуть використовуватися як тимчасові змінні.

Для зберігання наших даних в SRAM необхідно перш за все зробити розмітку в цій пам'яті під наші змінні за допомогою директиви .byte.

```
        .DSEG
foo:    .byte   1           ;резервує 1 байт для змінної foo
date:   .byte   3           ;0-день, 1-місяць, 2-останні дві цифри року
```

Для запису в SRAM в асемблері AVR передбачено 12 команд. Більшість з них є однотиповими та відрізняються лише використанням різних індексних регістрових пар X, Y, Z.

Найпростішою є команда sts, вона заносить у вказану адресу комірки пам'яті значення з регістра загального призначення.

```
        .CSEG
        ldi     r16, 53
        sts     foo, r16        ; $060← 53      запис у SRAM

        ldi     r16, 9
        sts     date, r16       ; $061← 9       запис у SRAM
        ldi     r16, 1
        sts     date+1, r16     ; $062← 1       запис у SRAM
        ldi     r16, 12
        sts     date+2, r16     ; $063← 12      запис у SRAM
```

Memory

Data | 8/16 | abc.

```
000060 053 009
000062 001 012
000064 000 000
```

Для безпосереднього зчитування даних з комірок SRAM є аналогічна команда lds.

```
lds     r16, foo        ; r16=53 ← $060  зчитування зі SRAM
lsl     r16             ; r16=53*2=106
subi    r16, 6          ; r16=106-6=100
sts     foo, r16        ; $060 ← 100      запис у SRAM
```

Решта асемблерні команди працюють зі SRAM через індексні регістрові пари X, Y, Z. Тобто, назви команд однакові: st (для запису) та ld (для зчитування), але аргументи різні : X, Rd; X+, Rd; -X, Rd; Y, Rd; Y+, Rd; -Y, Rd; Z, Rd; Z+, Rd; -Z, Rd (для команд зчитування аргументи розташовані навпаки). При використанні команд st та ld спершу у вибрану регістру пару заноситься адреса комірки пам'яті в SRAM. Далі, у залежності від алгоритму, використовується одна з трьох нотацій для вибраної регістрової пари (звичайна, з плюсом, з мінусом).

Команди зі звичайною нотацією працюють з комірками SRAM через адреси, що визначені в регістрових парах, наприклад, команда st X, Rd заносить значення з регістра загального призначення у комірку SRAM за адресою, що вказана в регістровій парі X.

Команди з плюсом після регістрової пари у нотації зчитують/записують дані з/у SRAM, а потім збільшують значення адреси в регістровій парі на одиницю.

Команди з мінусом перед регістровою парою у нотації спершу зменшують значення адреси в регістровій парі на одиницю, а потім зчитують/записують дані з/у SRAM.

Покажемо на тривіальному прикладі використання цих команд. Задача полягатиме у зчитуванні з порту А довільних 100 вибірок зі збереженням їх у SRAM та виводом їх на порт В у зворотному порядку (останнє збережене значення піде першим на вивід).

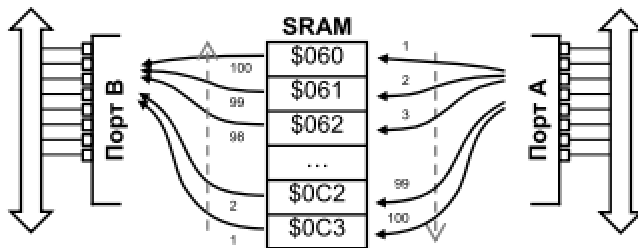


Рис. 4. Ілюстрація задачі зворотного вводу-виводу даних

```
.include "m32Adef.inc"
        .DSEG
data:   .byte   100             ; резервуємо 100 байт для наших даних

        .CSEG
        ldi     r16, 0x00
        ldi     r17, 0xFF

        out     DDRA, r16       ; порт A на вхід
        out     PORTA, r16

        out     DDRB, r17       ; порт B на вихід
        out     PORTB, r16
        clr     r20             ; r16=0
        ldi     ZL, low(data)   ; Z ← значення адреси мітки data
        ldi     ZH, high(data)

input:  in      r16, PINA       ; r16 ← порт A
        st      Z+, r16         ; SRAM ← r16;  (Z++)
        inc     r20             ; r16++
        cpi     r20, 100
        brne    input           ; if(r16!=100) перехід на мітку input

        clr     r20             ; r16=0
output: ld      r16, -Z         ; (Z--);  r16 ← SRAM
        out     PORTB, r16      ; порт B ← r16
        inc     r20             ; r16++
        cpi     r20, 100
        brne    output          ; if(r16!=100) перехід на мітку output
```

Також є ще дві корисні команди для роботи зі SRAM: std Y+k, Rr та ldd Rr, Y+k. Ці команди працюють лише із регістровими парами Y та Z, та виконують непрямий відносний запис/ зчитування у /з SRAM. Адреса комірки пам'яті, до якої ми звертаємося, отримується в результаті сумування значення регістрової пари та константи, при цьому значення регістрової пари не змінюється.

```
        ldi     ZL, low(date)
        ldi     ZH, high(date)  ; Z=$060 ← адреса мітки data

        ldi     r16, 25         ; r16=25
        std     Z+2, r16        ; $062 ← 25     (Z=$060)
        std     Z+5, r16        ; $065 ← 25     (Z=$060)
        ldd     r16, Z+10       ; r16 ← $06A    (Z=$060)
```

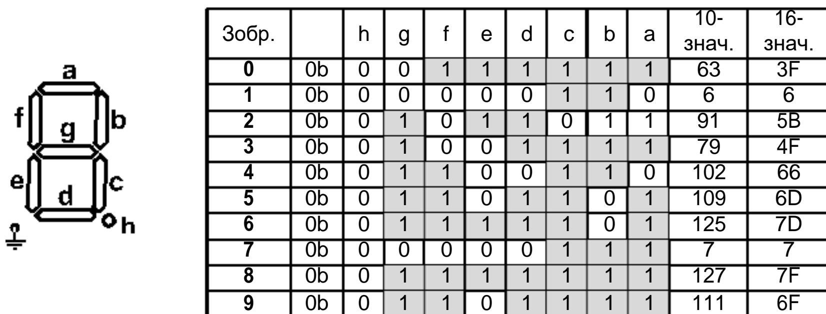
Команди для роботи зі SRAM виконується МК за 2 такти.

FLASH. Для реалізації деяких задач у програмі іноді виникає необхідність мати у наявності таблиці наперед визначених даних, наприклад, таблиця синусів, косинусів, вектори значень для двійково-десяткового кодування тощо. Найпростіше ці значні масиви даних зберігати безпосередньо в пам'яті програм, в області після коду програми, щоб наші таблиці з даними не перешкоджали його виконанню. Звичайно, ми можемо і посеред програми зробити острівок з даними та перестрибувати його за допомогою команд переходу.

Покажемо роботу з пам'яттю програм на конкретному прикладі, використовуючи вивчений вже нами матеріал. Реалізуємо простеньку задачу по циклічному виводу чисел від 1 до 99 з інтервалом в 0,5 сек. на семисегментні індикатори.

Семисегментний графічний індикатор складається зі світлодіодних сегментів. Засвічуючи певні сегменти, можемо вивести необхідне число. Семисегментні індикатори є зі спільним анодом (+) та спільним катодом (–). Це означає, що у такому індикаторі в усіх сегментах один
з виводів об'єднаний з рештою подібних виводів інших сегментів. І тоді на спільний вивід подається або високий рівень напруги (+), або низький (–), у залежності від конструкції вибраного індикатора. На решта виводів подають протилежні за рівнем напруги для засвічування сегментів.

Розглянемо індикатор зі спільним катодом. На спільний вивід подаємо низький рівень, а для засвічування сегментів необхідно з МК подавати високі рівні. На рис. 5 зображено вигляд індикатора з прийнятою літерною нумерацією сегментів, та наведена таблиця з відповідними кодами, які необхідно подати на індикатор, щоб засвітилася певна цифра. Наприклад, для висвічування цифри «5» необхідно засвітити сегменти a, f, g, c, d, тобто подати на них високий рівень.



Зобр.		h	g	f	e	d	c	b	a	10-знач.	16-знач.
0	0b	0	0	1	1	1	1	1	1	63	3F
1	0b	0	0	0	0	0	1	1	0	6	6
2	0b	0	1	0	1	1	0	1	1	91	5B
3	0b	0	1	0	0	1	1	1	1	79	4F
4	0b	0	1	1	0	0	1	1	0	102	66
5	0b	0	1	1	0	1	1	0	1	109	6D
6	0b	0	1	1	1	1	1	0	1	125	7D
7	0b	0	0	0	0	0	1	1	1	7	7
8	0b	0	1	1	1	1	1	1	1	127	7F
9	0b	0	1	1	0	1	1	1	1	111	6F

Рис. 5**.** Кодування семисегментного індикатора зі спільним катодом

Схему підключення семисегментних індикаторів до портів МК виконаємо, як на рис.6.

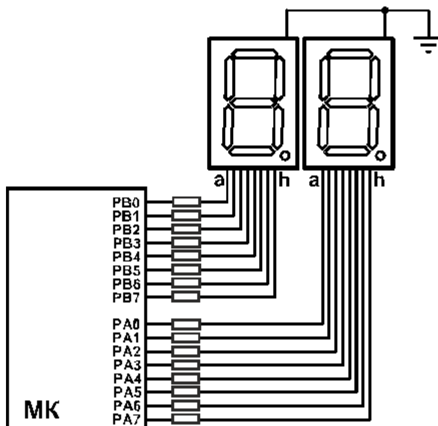


Рис. 6. Схема підключення індикаторів для задачі виводу числа

Як ми вже згадували, для розмітки простору у пам'яті FLASH використовуються такі директиви: .db (байт), .dw (слово або 2 байти), .dd (2 слова або 4 байти) та .dq (4 слова або 8 байт). Для нашої задачі нам необхідно створити таблицю з 10-ти однобайтних значень, які б відповідали кодам виводу від 0 до 9 для нашого семисегментного індикатора. Значення розміщуються у таблиці у порядку зростання відображуваного числа. Тобто, якщо нам необхідно буде вивести число «5», тоді ми до адреси мітки таблиці додамо значення зсуву 5 і отримаємо необхідний код.

Зчитування значення з пам'яті FLASH виконується командою lpm (3 такти) згідно адреси, що вказана у регістровій парі Z. Тобто, нам спершу необхідно загрузити в Z адресу мітки таблиці, а потім до Z додати необхідне зміщення у таблиці та виконати команду зчитування.

```
.include "m32Adef.inc"
.def     _8bin    =r18
.def     _bcdL    =r18
.def     _bcdH    =r19
; Константи
.equ     Fig_0 = 63         ; 0
.equ     Fig_1 = 6          ; 1
.equ     Fig_2 = 91         ; 2
.equ     Fig_3 = 79         ; 3
.equ     Fig_4 = 102        ; 4
```

```
.equ    Fig_5 = 109        ; 5
.equ    Fig_6 = 125        ; 6
.equ    Fig_7 = 7          ; 7
.equ    Fig_8 = 127        ; 8
.equ    Fig_9 = 111        ; 9

        .CSEG
        ldi     r16, Low(RAMEND)     ┐
        out     SPL, r16             │
                                     ├ Ініціалізація стеку
        ldi     r16, High(RAMEND)    │
        out     SPH, r16             ┘

        ldi     r16, 0x00
        ldi     r17, 0xFF
        out     DDRA, r17          ; порт А на вихід
        out     PORTA, r16
        out     DDRB, r17          ; порт В на вихід
        out     PORTB, r16

        clr     r20                ; r20=0 (число для виводу на індикатори)

main:   mov     _8bin, r20         ; r18 ← r20
;----------    Виклик підпрограми 2-10 кодування ---------------------------
        rcall   BCD    ; вхід: _8bin;      вихід: _bcdL(мол.), _bcdH(старш.)

;----------    Вивід молодшого розряду десяткового числа ---------------
        ldi     ZL, low(SegTable*2)
        ldi     ZH, high(SegTable*2)   ;Z ← адреса мітки SegTable у байтах
        clr     r16
        add     ZL, _bcdL
        adc     ZH, r16                ; Z = Z + _bcdL (зміщення)
        lpm     r16, Z                 ; r16 ← FLASH(Z)
        out     PORTA, r16             ; порт А ← r16

;----------    Вивід старшого розряду десяткового числа -----------------
        ldi     ZL, low(SegTable*2)
        ldi     ZH, high(SegTable*2)   ;Z ← адреса мітки SegTable у байтах
        clr     r16
        add     ZL, _bcdH
        adc     ZH, r16                ; Z = Z + _bcdH (зміщення)
        lpm     r16, Z                 ; r16 ← FLASH(Z)
        out     PORTB, r16             ; порт В ← r16

;----------    Виклик підпрограми затримки -----------------------------------
        rcall   P05sec
;----------    Інкрементування числа r20 до 99 ----------------------------------
        inc     r20                ; r20++
        cpi     r20, 100
        brne    main               ; if (r20 != 100) goto main
        clr     r20                ; else  r20=0; goto main
        rjmp    main
```

```
;Підпрограма Двійково-Десяткового Кодування (до 99, незапаковане)
BCD:     clr     _bcdH
BCD1:    subi    _bcdL, 10
         brcs    BCD2
         inc     _bcdH
         rjmp    BCD1
BCD2:    subi    _bcdL, -10
         ret

;Підпрограма паузи 0.5 сек.
P05sec:  ldi     r16, 0x00
         ldi     r17, 0x35
         ldi     r18, 0x0C
delay:   subi    r16, 1
         sbci    r17, 0
         sbci    r18, 0
         brne    delay
         ret

;Вектор даних
SegTable: .db    Fig_0,Fig_1,Fig_2,Fig_3,Fig_4,Fig_5,Fig_6,Fig_7,Fig_8,Fig_9
      ;коди цифр                  0123456789
```

Частини коду у програмі, що мають відношення до роботи з пам'яттю FLASH, відмічені заокругленими прямокутниками:

– розміщення таблиці даних у пам'яті програм з використанням директиви .db ;

– читання необхідного значення з пам'яті за адресою, що розміщена в регістровій парі Z.

EEPROM. Ця пам'ять, аналогічно SRAM, розташована у своєму адресному просторі та організована лінійно. Доступ до пам'яті побайтний. Для роботи з EEPROM використовуються 3 регістри вводу/виводу:

EEAR – регістр адреси EEPROM-пам'яті, який фізично розташовується у двох регістрах EEARH: EEARL. У цей регістр завантажується адреса комірка, до якої буде звертатися звертання, як для запису, так і для читання.

EEDR – регістр даних. У нього завантажуються дані для запису в EEPROM, а також у ньому розміщуються дані, отримані при читанні з EEPROM-пам'яті.

EECR – регістр керування доступом до EEPROM-пам'яті. У ньому задіяні 4 біти для визначення поведінки МК при роботі з EEPROM.

3	EERIE	Дозвіл на переривання по завершенню запису в EEPROM
2	EEMWE	Попередній дозвіл на запис даних. Після нього повинен одразу бути встановлений біт EEWE, інакше він буде апаратно скинутий через 4 такти.
1	EEWE	Дозвіл на запис. При встановленні в 1 відбувається запис даних у EEPROM (при умові, що встановлений біт EEMWE).
0	EERE	Дозвіл на читання**.** При встановленні в 1 відбувається читання з EEPROM. По завершенню читання цей розряд скидається апаратно.

Алгоритм запису одного байту в EEPROM-пам'ять:

1. Дочекатися готовності EEPROM (поки не скинеться біт EEWE).
2. Завантажити байт даних у регістр EEDR, а необхідну адресу у регістр EEAR.
3. Встановити в «1» біт EEMWE регістра керування EECR.
4. Записати у розряд EEWE регістра керування логічну «1» протягом 4-х тактів.

Тривалість запису в EEPROM є дуже великою та складає приблизно 2-9 мсек. Тому послідовний запис деякого числа байтів в EEPROM можемо організувати через проміжний буфер FIFO. Тобто, ми запишемо дані усі підряд в буфер, а він, використовуючи процедуру переривання по завершенню запису, буде по байту записувати наші дані в EEPROM, працюючи при цьому у фоновому режимі. Про організацію таких буферів ми розкажемо при розгляді роботи послідовного порту USART.

Ще один необхідно звернути увагу на те, що якщо між пп .3 та 4 відбудеться переривання, то зірветься уся процедура запису. Тому необхідно на цей час забороняти будь-які переривання.

Процедура читання має подібний алгоритм: очікуємо готовності EEPROM, далі завантажуємо необхідну адресу у регістр EEAR, встановлюємо в «1» біт дозволу на читання EERE та зчитуємо наш байт даних з регістра EEDR. Читання відбувається за 1 такт, однак прогальмовування у 4 такти відбувається при встановленні біта дозволу на читання EERE.

Наведемо приклад програми для роботи з EEPROM. Записане у EEPROM, а потім зчитане назад з нього число виводимо на лінійку зі світлодіодів, що підключені до порту А (рис. 7).

```
.include "m32Adef.inc"
          .CSEG
          ldi     r16, 0x00
          ldi     r17, 0xFF
          out     DDRA, r17              ; порт А на вихід
          out     PORTA, r16
          ldi     r18, 159               ; r18=0b10011111
;---------- запис в E_Pass -----------------------------
E_Write:  sbic    EECR, EEWE             ; очікуємо готовності EEPROM
          rjmp    E_Write
          ldi     r17, high(E_Pass)      ; отримуємо адресу мітки E_Pass
          ldi     r16, low(E_Pass)
          out     EEARH, r17             ; завантажуємо адресу в рег. EEAR
          out     EEARL, r16
          out     EEDR, r18              ; заносимо байт даних для запису
          cli
          sbi     EECR, EEMWE            ; вст. попередній дозвіл на запис
          sbi     EECR, EEWE             ; вст. остаточний дозвіл на запис
          sei
;---------- читання з E_Pass ----------------------------
E_Read:   sbic    EECR, EEWE             ; очікуємо готовності EEPROM
          rjmp    E_Read
          ldi     r17, high(E_Pass)      ; отримуємо адресу мітки E_Pass
          ldi     r16, low(E_Pass)
          out     EEARH, r17             ; завантажуємо адресу в рег. EEAR
          out     EEARL, r16
          sbi     EECR, EERE             ; вст. дозвіл на читання
          in      r20, EEDR              ; зчитуємо байт з рег. EEDR
          out     PORTA, r20             ; виводимо байт на світлодіоди

main:     rjmp    main

;розмітка пам'яті EEPROM
          .ESEG
E_Pass:   .db     145     ;0b10010001
```

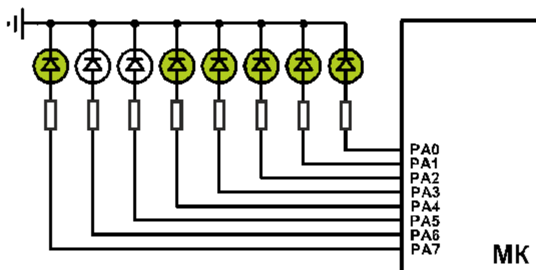


Рис. 7. Результат виконання програми при роботі з EEPROM

Розмітку областей EEPROM-пам’яті можемо виконувати, як у пам’яті програм, за допомогою директив .db, .dw, .dd, .dq, що вказують на розміщення даних, так і за допомогою директиви .byte, що резервує необхідну кількість байтів (як у SRAM).