Лекція 1. Варіанти структур цифрових керуючих систем

МЕТА ЛЕКЦІЇ — вивчити типові структури систем управління з мік- роЕОМ, способи з'єднання мікроконтролерів, ознайомитися з основними поняттями мікроконтролерного управління.

Розглянемо такі питання:

1. Варіанти організації мікроконтролерного управління.
2. Показники надійності й вартості при виборі варіантів систем.

Варіанти організації мікроконтролерного управління

Значні обчислюв&іьні й логічні можливості мікропроцесорів і систем на їхній основі визначають доцільність їх використання для автоматичного й автоматизованого (за участю людини) управління об'єктами. Мікро- процесор, як і будь-які інші пристрої цифрової обробки сигналів, має важливі переваги перед аналоговими пристроями. Це висока стабільність характеристик, відсутність дрейфу нуля, висока точність виконання арифметичних операцій, невеликі вага і габарити, висока швидкодія, можливість гнучкої оперативної перебудови структури та ін.

Крім того, багато фізичних систем є дискретними, тобто їхнє по- водження може бути описане дискретними чи цифровими моделями. Наприклад, у радарних системах передані й прийняті сигнали є імпу- льсними. Існують численні явища, соціальні, економічні й біологічні системи, динаміка яких може бути описана дискретними моделями.

Цифрові контролери порівняно з аналоговими регуляторами мають такі переваги, як підвищена чутливість, велика надійність, більш висока стійкість до шумів і збуджень, зручність у програмуванні, менша вартість.

Програма цифрового регулятора може бути змінена відповідно до вимог проектувальників або пристосована до характеристик об'єкта без будь-яких змін в апаратному забезпеченні. Цифрові компоненти електронних схем надійніші, міцніші й компактніші, ніж аналогові компоненти того самого призначення.

Нашим часом при створенні систем автоматичного управління принципово можливо йти по двох напрямах. Перший з них зв'язаний з використанням центральних мікроЕОМ, що управляють. Подібні системи можуть застосовуватися для управління складними об'єктами (літаками, ракетами, прокатними станами, доменними печами тощо)

^« . ·u ... .

чи групами оо єктів при комплексній автоматизації в різних галузях

промисловості й сільського господарства (металургійній, хімічній, нафтопереробній, у тепличному овочівництві, тваринництві та ін.).

Загальна структурна схема автоматичної системи з мікроЕОМ для цього разу показана на рис. 1.1. Система містить ряд вхідних *ВхП*$_1$, *ВхП*$_2$,..., *ВхП*$_n$ та вихідних *ВихП*$_1$, *ВихП*$_2$,..., *ВихП*$_n$ перетворювачів, що обмінюються сигналами управління з мікроЕОМ по шині управління у процесі перетворення даних. Сигнали з вихідних перетворювачів надходять на виконавчі пристрої *ВП*$_1$, *ВП*$_2$, ..., *ВП*$_n$, що впливають на об'єкт (чи об'єкти) управління.

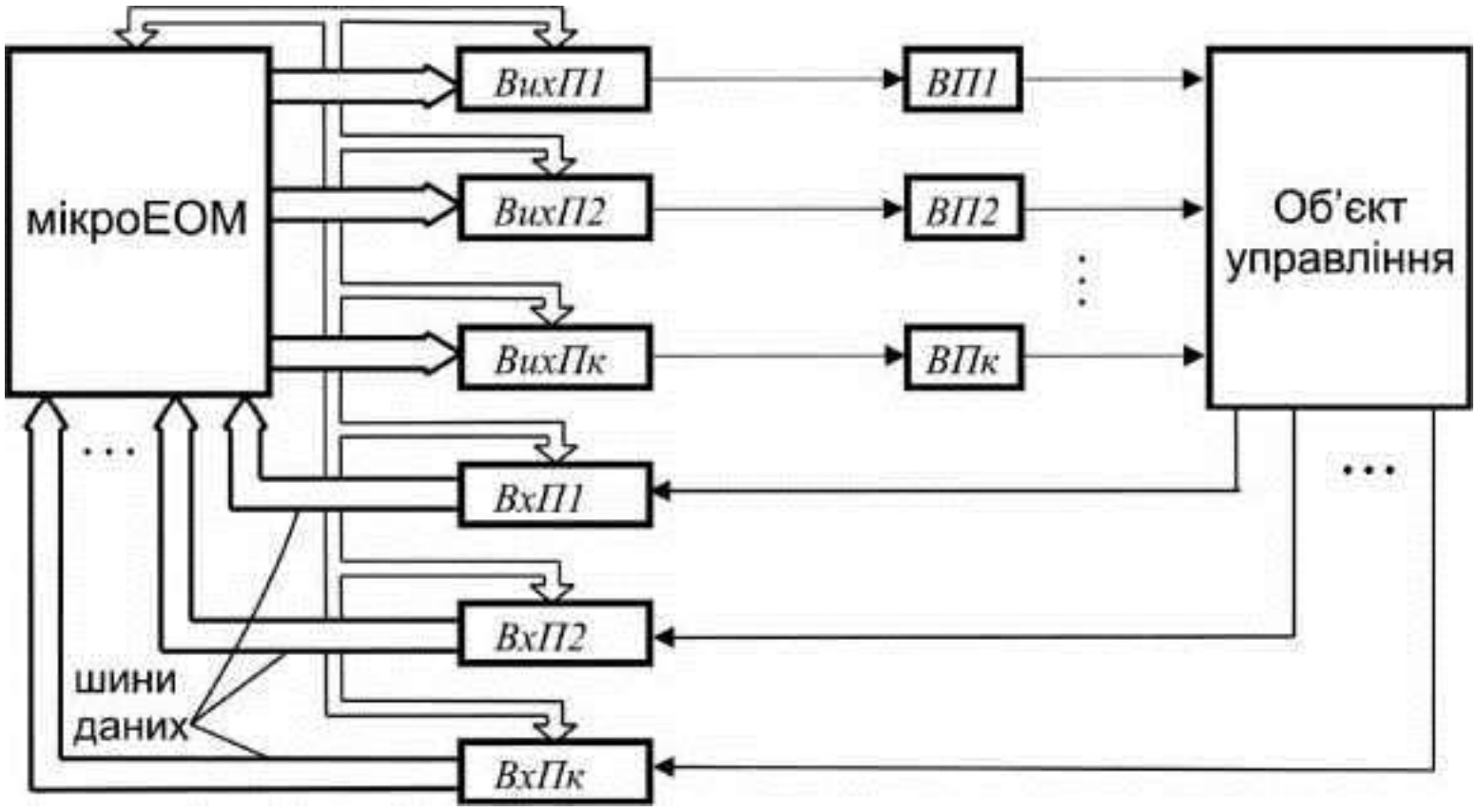


Рис. 1.1. Структурна схема системи управління з центральною мікроЕОМ, що управляє

Така система може бути зв’язаною багатовимірною, якщо здійснюється управління складним багатовимірним об’єктом, і незв'язаною багатовимірною, якщо здійснюється управління групою незв’язаних одновимірних об’єктів. В останньому разі САУ з мікроЕОМ розпадається на сукупність одновимірних систем; її віртуальна (удавана) структура надається у вигляді сукупності одноконтурних систем автоматичного управління (рис. 1.2), кожна з яких має свою програму управління $ПУ_y П У П У_к$.

При управлінні складним об'єктом чи групою об'єктів процесор обслуговує по черзі окремі канали управління.

Ця черга може здійснюватися за програмою чи в міру надходження заявок від окремих каналів з можливістю використання в останньому разі пріоритетного обслуговування.

Другий напрям, за яким розвиваються даним часом САУ з мікро- ЕОМ, — це використання в кожнім контурі управління автономної мікроЕОМ, яка називається мікроконтролером.

Структурна схема САУ з автономними мікроконтролерами $МК_x$, $МК_2$,..., $МК_к$ показана на рис. 1.3.

Мікроконтролери являють собою спрощені варіанти мікроЕОМ, що розташовані в безпосередній близькості від об’єкта управління. У мікроконтролерних системах центральна ЕОМ або відсутня зовсім, або вводиться для передачі їй функцій диспетчера чи супервізора.

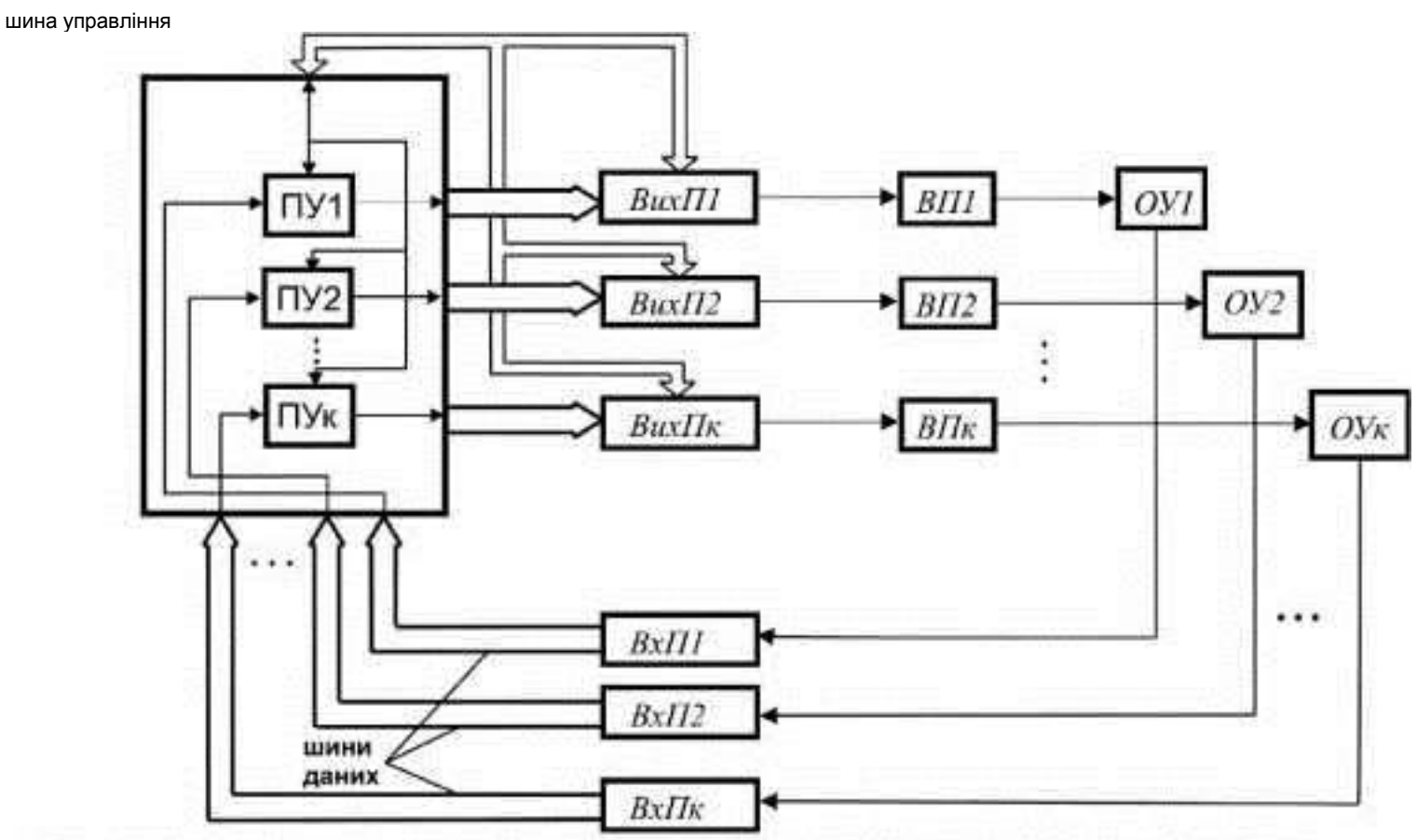


Рис. 1.2. Віртуальна структура САУ з центральною мікроЕОМ

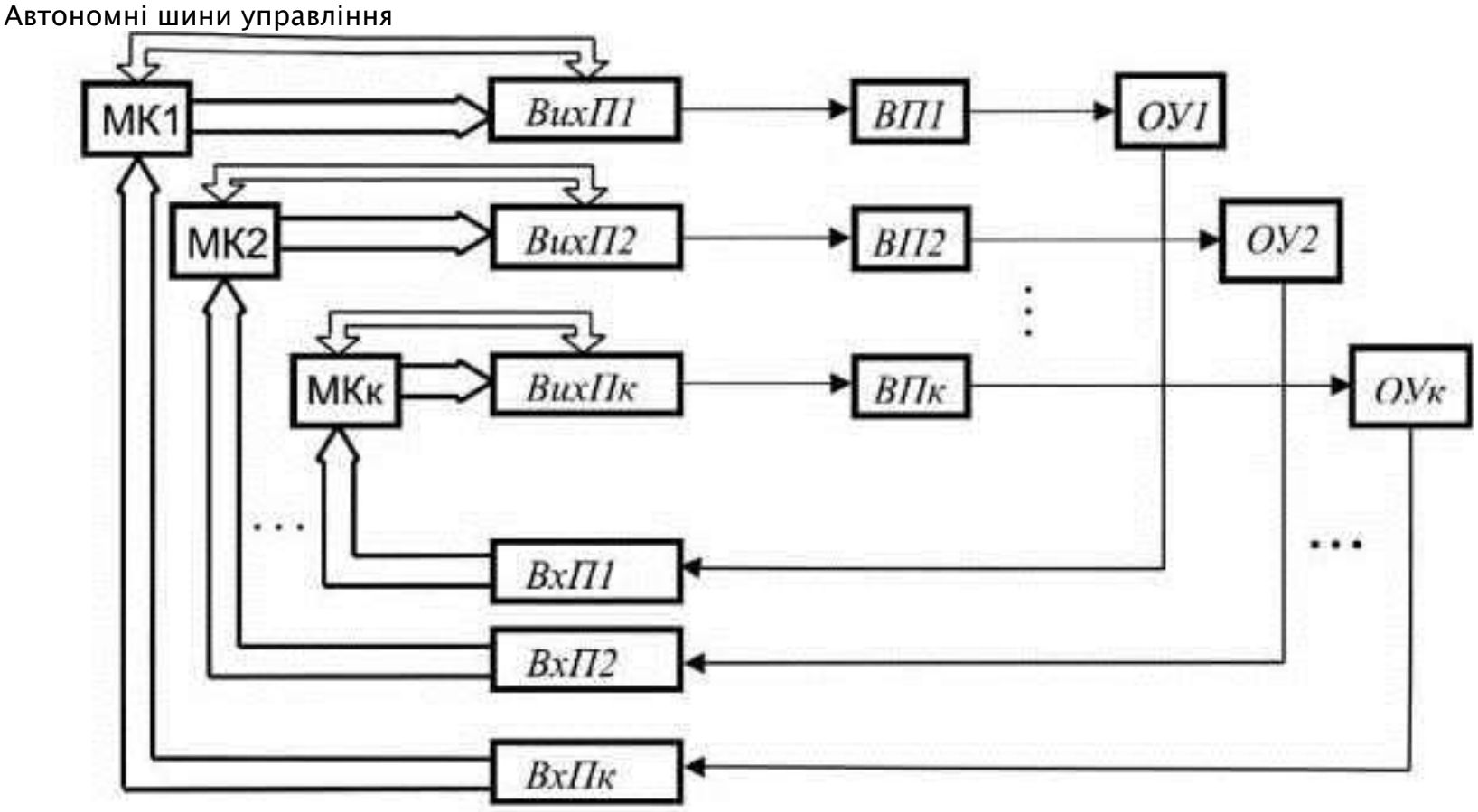


Рис. 1.3. Система з автономними мікроЕОМ, що управляють

Показники надійності й вартості при виборі варіантів систем

Вибір одного з двох напрямів побудови САУ з мікроЕОМ зв'язаний, зокрема, із проблемами надійності й вартості. Вартість систем, які використовують центральну мікроЕОМ, шо управляє, звичайно при великому числі керованих об'єктів нижче вартості мікроконт- ролерних систем (рис. 1.4, де $K_ц$ — вартість систем з центральною мікроЕОМ: K — вартість мікроконгролерних систем). Ця законо- мірність з розвитком технології виробництва мікропроцесорів, ЩО привела до створення високоефективних однокристальних мікро- ЕОМ, виявляється дедалі меншою мірою. Крім того, системи на базі центральних мікроЕОМ, що управляють, є технологічно менш надійними (рис. 1.5). Вони мають потребу в дорогих, перешкодос- тійких лініях зв'язку. Тому принцип децентралізованого (мікроконтролерного) управління в мікропроцесорних системах поступово стає переважним.

Мікроконтролерне управління з позицій теорії надійності може бути організоване (рис. 1.6) одним із таких способів: а) управління з конвеєрною (послідовною) обробкою інформації; б) управління па- ралельною обробкою інформації: в) мажоритарне управління.

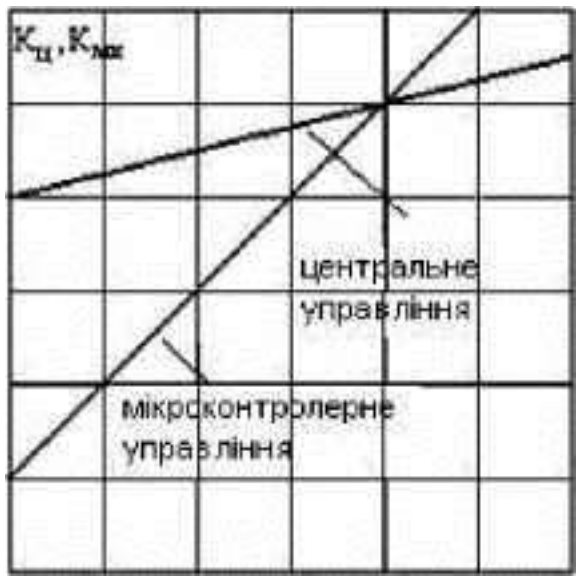


Рис. 1.4. Залежність відносної зміни вартості мікропроцесорних засобів у САУ від числа об'єктів управління

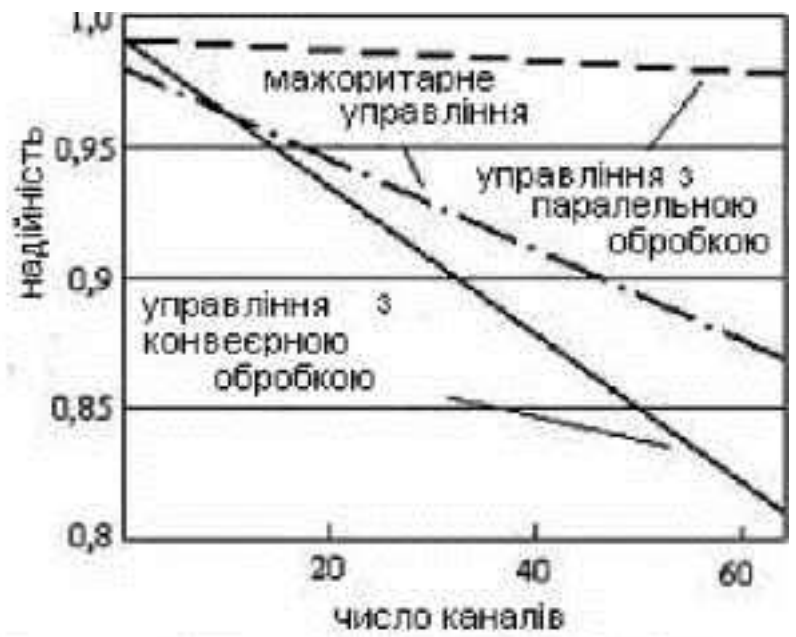
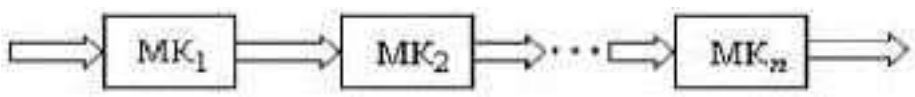


Рис. 1.5. Залежність надійності САУ з мікроЕОМ від числа каналів управління



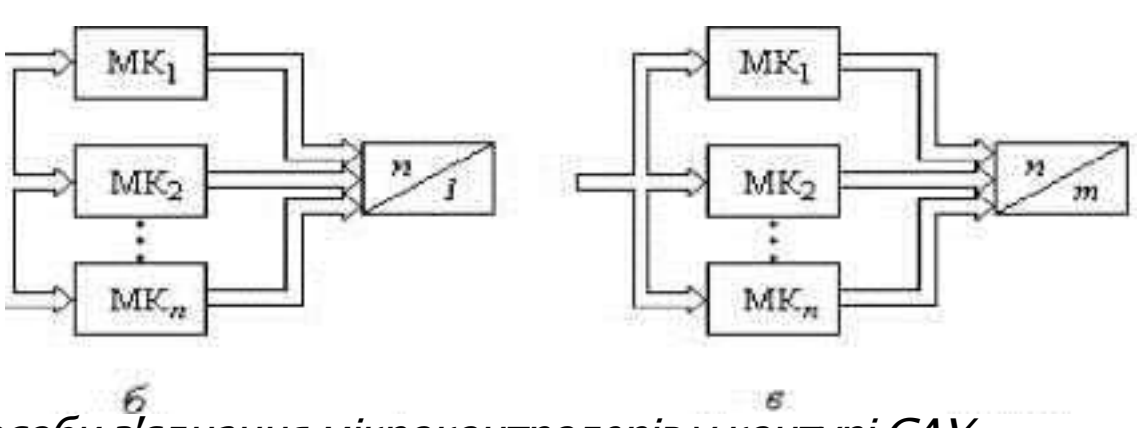


Рис. 1.6. Способи з'єднання мікроконтролерів у контурі САУ

Запитання для самоконтролю

1. Структури цифрових систем управління і призначення їх блоків.
2. Напрями розвитку мікропроцесорних САУ.
3. Принцип децентралізованого мікропроцесорного управління.
4. Порівняйте характеристики цифрових і аналогових регуляторів.
5. Способи з'єднання мікроконтролерів у контурі САУ.

Література:

1. Бесекерский В. А., Изранцев В. В. Системы автоматического управлення с микроЗВМ. — М.: Наука, 1990. — 320 с.
2. Великий В.І., Препелиця Г. П. Мікропроцесорні системи обробки даних та управління в гідрометеорології: Навч. посіб. — О.: Ви-во ТЗС, 2004.- 212 с.
3. Локазюк В. М. Мікропроцесори та мікроЕОМ у виробничих системах: Навч. посіб. для вузів. — К.: Академія, 2002. — 368 с.
4. Микрогіроцессорное управление злектроприводами станков с ЧПУ/(З. Л. Тихомиров, В. В. Васильєв, Б. Г. Коровин, В. А. Яков- лев). — М.: Машиностроение, 1990. — 319 с.
5. Микроироцессорньїесистемьі: Учеб. пособиедля вузов/Е. К. Алек- сандров, Р. И. Грушвицкий, М. С. Куприянов и др.; Под обш. ред. Д. В. Пузанкова. — СПб.: Политехника, 2002. — 935 с.: ил.

Лекція 2. Мікроконтролери AVR

Мета лекції: ознайомитись зі структурою і характерними особливостями мікроконтролерів AVR фірми ATMEL.

Питання:

Огляд мікроконтролерів AVR фірми ATMEL.

Адресація регістрів введення/виведення й пам'яті SRAM.

Програмний лічильник і стек.

Регістр стану. Переривання.

Загальна характеристика Atmega16 і робота с портами.

Огляд мікроконтролерів AVR фірми ATMEL.

Однокристальні мікроконтролери знаходять широке застосування в найрізноманітніших сферах: від вимірювальних приладів, фотоапаратів і відеокамер, принтерів, сканерів і копіювальних апаратів до виробів електронних розваг і всілякої домашньої техніки.

Із часу появи перших мікропроцесорів (1970-і рр.) їх складність постійно зростала у зв'язку з появою нових апаратних рішень і додаванням нових команд, необхідних при виконанні нових завдань. Так поступово склалась архітектура, що одержала назву CISC (Complex Instruction Set Computers - комп'ютери зі складним набором команд). Надалі позначилося й знайшов активний розвиток ще один напрямок: архітектура RISC (Reduced Instruction Set Computers - комп'ютери зі скороченим набором команд). Саме до цієї архітектури відносяться мікроконтролери AVR від компанії.

Основна перевага RISC-процесорів - вони прості, виконують обмежений набір команд, що приводить до швидкодії швидкості операції. Це дозволяє знизити вартість і складність їх програмування.

На зорі виникнення мікропроцесорів розробка програмного забезпечення відбувалася винятково на тій або іншій мові асемблера, орієнтованому на конкретний пристрій. По суті, такі мови являли собою символьні мнемоніки відповідних машинних кодів, а переклад мнемоніки в машинний код виконувався транслятором. При цьому головний недолік ассемблерных мов є в тім, що кожний з них був прив'язаний до конкретного типу пристроїв і логіку його роботи. Крім того, асемблер складний в освоєні, що вимагає досить більших зусиль для його вивчення, але головне, якщо згодом буде потрібно перейти на використання мікроконтролерів інших виробників, то зусилля виявляться марними.

Мова С, будучи мовою високого рівня, позбавлена подібних ндоліків і може використатися для програмування кожного мікропроцесора, для якого є компілятор з мови С. Вивчивши мову Сі, можна легко переходити від одного сімейства мікроконтролерів до іншому, витрачаючи набагато менше часу на разробку.

1. Архітектура AVR мікроконтролерів AVR фірми ATMEL.

Для досягнення дуже швидкого й ефективного виконання про- грам архітектура AVR була оптимізована таким чином, щоб зкомпонувати переваги Гарвардської й Прінстонської архітектури. Така організація забезпечує високу ефективність процесора при обробці даних.

Основною ідеєю всіх RISC (Reduced Instruction Set Computer) є збільшення швидкодії за рахунок скорочення кількості операцій обміну з пам'яттю програм. Для цього кожну команду прагнуть вмістити в одну комірку пам'яті програм. При обмеженый розрядності комірки пам'яті це неминуче приводить до скорочення набору команд мікропроцесора [1].

Тому в AVR-микроконтроллеров відповідно до цього принципу практично всі команди (крім тих, у яких одним з операндыв є 16-розрядна адреса) також упаковані в одну комірку пам'яті програм. Однак зробити це вдалося не за рахунок скорочення кількості команд процесора, а розширенням комірки пам'яті програм до 16 розрядів. Таке рішення дає можливість використати безліч команд AVR на відміну від інших RISC-микроконтроллерів.

Організація пам'яті AVR виконана за схемою Гарвардського типу, у якій розділені не тільки адресні простори пам'яті програм і пам'яті даних, але також і шини доступу до них [2].

Вся програмна пам'ять AVR-микроконтроллерів виконана по технології FLASH і розміщена на кристалі. Вона являє собою послідовність 16-розрядних комірок і має ємність від 512 слів до 256K слів залежно від типу кристала.

Поділ шин доступу (рис. 1.1) до FLASH пам'яті й SRAM пам'яті дає можливість мати шини даних для пам'яті даних і пам'яті програм різної розрядності, а також використати технологію конвейеризації. Конвеєризація полягає в тім, що під час виконання поточної команди програмний код наступної вже вибирається з пам'яті й дешифрується [1].

Для порівняння згадаємо, що в мікроконтролерів сімейства MCS-51 вибірка коду команди і її виконання здійснюються послідовно, а це займає один машинний цикл, що триває 12 періодів кварцового резонатора [2].

У випадку використання конвеєризації тривалість машинного циклу можна скоротити. Наприклад, у мікроконтролерів фірми Microchip (PIC) завдяки використанню конвеєра вдалося зменшити тривалість машинного циклу до чотирьох періодів кварцового резонатора [3]. Тривалість же машинного циклу AVR становить один період кварцового резонатора, тому AVR-микроконтролери здатні забезпечити задану продуктивність при більше низькій тактовій частоті. Саме ця особливість архітектури й дозволяє AVR-мікроконтролерам мати найкраще співвідношення енергоспоживання/производительность, тому що споживання КМОП мікросхем визначається їхньою робочою частотою [1].

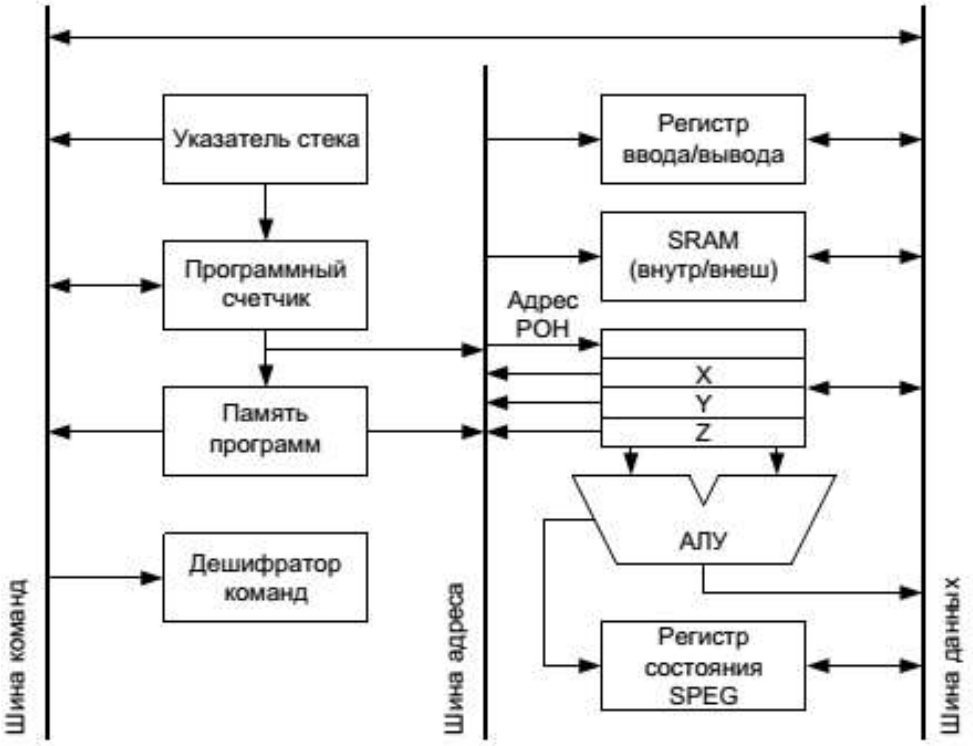


Рис. 1.1. Структурна схема архітектури процесора сімейства AVR

1.1.1. Програмна модель AVR і система команд

На рис. 1.2 зображена програмна модель AVR-контролера, яка являє собою діаграму програмно доступних ресурсів. Центральним блоком на цій діаграмі є регістровий файл із 32 оперативних регістрів (R0-R31), або (як їх звичайно називають) регістрів загального призначення (РЗП). Всі РЗП безпосередньо доступні арифметико-логічному пристрою (АЛП). Старші регістри об'єднані парами (мал. 1.3.) і утворюють три 16-розрядних регістри, для непрямої адресації комірок пам'яті (AVR без SRAM мают тільки один 16-бітний регістр) [1].

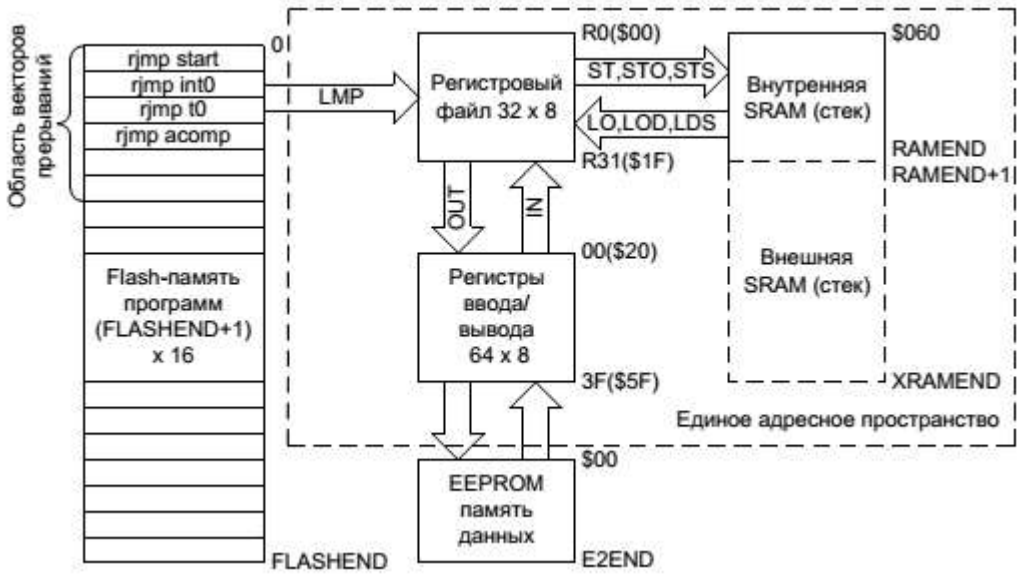


Рис. 1.2. Програмна модель AVR-мікроконтролера

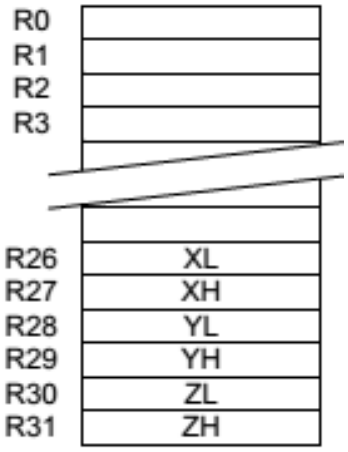


Рис. 1.3. Регістровий файл або РЗП

Регістровий файл, блок регістрів введення/виведення й оперативна пам'ять (див. мал. 1.2) утворюють єдиний адресний простір, що дає можливість при програмуванні звертатися до 32 оперативних регістрів і до регістрів введення/виведення як до комірок пам'яті, використовуючи команди доступу до SRAM (у тому числі й з непрямою адресацією).

Всі арифметичні й логічні операції, а також частина операцій роботи з бітами виконуються в АЛП тільки над умістом РЗП. Варто пам'ятати, що команди, які в якості другого операнда мають константу (SUBI, SBCI, ANDI, ORI, SBR, CBR), можуть використати в якості першого операнда тільки регістри із другої половини РЗП.(R16-R31). Команди 16-розрядного додавання з константою ADIW і віднімання константи SBIW у якості першого операнда використають тільки регістри R24, R26, R28, R30.

Під час виконання арифметичних і логічних операцій або операцій роботи з бітами АЛП формує ті або інші ознаки результату операції, тобто встановлює або скидає біти в регістрі стану SREG (Status Register), які будуть розглянуті окремо.

Ознаки результату операції можуть потім використатися в програмі для виконання подальших арифметико-логічних операцій або команд умовних переходів [1, 2].

Адресація регістрів введення/виведення й пам'яті SRAM

Регістри введення/виведення являють собою набір регістрів управ- ління процесорного ядра й даних апаратних вузлів AVR-мікро- контролера. Регістрами введення/виведення є регістри SREG, MCUSR і покажчик стека SPH:SPL, а також регістри, що управляють системою переривання мікроконтролера, режимами підключення EEPROM пам'яті, сторожовим таймером, портами введення/виведення й дру- гими периферійними вузлами [1].

Всі регістри введення/виведення можуть зчитуватися й записуватися через РЗП за допомогою команд IN, OUT. Регістри введення/виведення, що мають адреси в діапазоні $00 - $1F (знак $ указує на шістнадцяткову систему числення), мають можливість побітової адресації. Безпосередня установка й скидання окремих розрядів цих регістрів виконуються командами SBI і CBI. Для ознак

результату операції, які є бітами регістра введення/виведення SREG, є цілий набор команд установки й скидання. Команди умовних переходів у якості своїх операндов можуть мати як біти-ознаки результату операції, так і окремі розряди побітно адресуємих регістрів введення/виведення.

Раніше було показано (мал. 1.2) розподіл адрес у єдиному адресному просторі. Молодші 32 адреси ($0 - $1F) відповідають оперативним регістрам, тобто РЗП. Наступної 64 адреси ($20 - $5F) зарезервовані для регістрів введення/виведення. Внутрішня SRAM у всіх AVR починається з адреси $60.

Із цього треба, що регістри введення/виведення мають подвійну нумерацію. Якщо використаються команди IN, OUT, SBI, CBI, SBIC, SBIS, то варто застосовувати нумерацію регістрів введення/виведення, що починається з нуля (назвемо її основною). Якщо ж до регістрів введення/виведення доступ здійснюється як до комірок пам'яті, то необхідно використати нумерацію єдиного адресного простору оперативної пам'яті даних AVR. Очевидно, що адреса в єдиному адресному просторі пам'яті даних виходить шляхом додатка числа $20 до основної адреси регістру введення/виведення [2, 3].

Для зберігання оперативних даних програміст, крім РЗП, може використати внутрішні й зовнішні (якщо вони є) блоки SRAM (мал. 1.2). Робота із зовнішньою SRAM може бути програмно дозволена/заборонена установкою/скиданням біта SRE у регістрі введення/виведення MCUSR.

Операції обміну із внутрішньою оперативною пам'яттю AVR-мікроконтролер виконує за два машинних цикли. Доступ до зовнішнього SRAM вимагає одного додаткового циклу на кожний байт у порівнянні із внутрішньою пам'яттю. Крім того, установкою біта SRW у регістрі введення/виведення MCUSR можна програмно збільшити час обміну із зовніньою SRAM ще на один додатковий машинний цикл очікування.

Виконувати арифметико-логічні операції й операції зсуву безпосередньо над вмістом комірок пам'яті не можна. Не можна також записати константу або очистити вміст комірки пам'яті. Система ко манд AVR дозволяє лише виконувати операції обміну даними між комірками SRAM і оперативними регістрами. Перевагою системи команд можна вважати різноманітні режими адресації комірок пам'яті. Крім прямої адресації є наступні режими: непряма, косвена з постінкрементом, непряма із предекрементном і непряма зі зсувом.

Оскільки внутрішн і зовнішня SRAM входять у єдиний адресний простір (разом з оперативними регістрами й регістрами введення/виведення), то для доступу до комірок внутрішньої й зовнішньої пам'яті використовуються ті самі команди.

Слід зазначити, що регістри введення/виведення не повністю використовують відведені для них 64 адреси. Невикористовувані адреси зарезервани для майбутніх застосувань, додаткових комірок пам'яті по цих адресах не існує [1].

3Програмний лічильник і стек

В комірках оперативної пам'яті організується системний стек, який використається автоматично для зберігання адрес повернення при виконанні підпрограм, а також може використатися програмістом для тимчасового зберігання вмісту оперативних регістрів (команди PUSH і POP). На початку будь-якої програми необхідно ініціалізувати стек програмними засобами, тобто

занести в Покажчик Стека (Stack Pointer) початкове значення, рівне самому старшому адресу комірки в оперативній пам'яті. Мікроконтролери, що не мають SRAM, містять трьохрівневий апаратний стек [1, 2].

Варто мати на увазі, що якщо стек розташовується в зовнішній SRAM, то виклики підпрограм і повернення з них вимагають двох додаткових циклів, якщо біт SRW не встановлений, і чотирьох, якщо встановлено.

Розмір стека в оперативній пам'яті, обмежений лише розмірами цієї пам'яті. Якщо мікроконтролер містить на кристалі 128 байт внутрішньої SRAM і не має можливості підключення зовнішній SRAM, то як покажчик вершини стека використається регістр введення/виведення SPL. Якщо є можливість підключення зовнішньої пам’яті або внутрішня пам'ять має розміри 256 байт і більше, то вказівник стека складається із двох регістрів введення/виведення SPL і SPH.

При занесенні числа в стек автоматично виконуються наступні дії:

- число записується в комірку пам'яті за адресою, що зберігається в покажчику стека (SPH:SPL) число;

/уміст покажчика стека зменшується на одиницю. SPH : SPL = = SPH : SPL - 1.

Зворотні дії виконуються при витягу числа зі стека:

- уміст покажчика збільшується на одиницю. SPH : SPL = = SPH : SPL + 1;
- число витягає з комірки пам'яті з адресою, що зберігається в ука- затілі стека (SPH : SPL) число. →

Таким чином, стек росте від старших адрес до молодшого. Тому, беручи до уваги, що початкове значення покажчика стека після скидання дорівнює нулю, програміст AVR обов'язково повинен в ініціаліующій частини програми подбати про установку покажчика стека, якщо він припускає використати хоча б одну підпрограму.

Крім оперативної пам'яті програмно-доступними ресурсами мік- роконтролера є енергонезалежні, електрично програмуємі FLASH і EEPROM блоки пам'яті, які мають окремі аресні простори.

Молодші адреси пам'яті програм мають спеціальне призначення. Адреса $0000 є адресою, з якого починає виконуватися програма після скидання процесора. Комірки пам'яті програм, починаючи з наступної адреси $0001, утворять область векторів переривання. У цій області для кожного можливого джерела переривання відведене свій адрес, по якому (у випадку використання даного переривання) розміщують команду відносного переходу RJMP на підпрограму обробки переривання (див. мал. 1.2). Варто пам'ятати, що адреси векторів переривання тих самих апаратних вузлів для різних типів AVR можуть мати й різне значення. Тому при переносі програмного забезпечення зручніше, так само як і у випадку з регістрами введення/виведення, використовувати символічні імена адрес векторів переривання.

EEPROM блок пам'яті, що стирається электрично, використовується для зберігання енергонезалежних даних, які можуть змінюватися безпосередньо на об'єкті. Це калібровані коефиціенти, різні установки, конфігураційні параметри

системи й т.д. EEPROM-пам'ять даних може бути програмним шляхом як лічена, так і записана. Однак спеціальних команд звертання до EEPROM немає. Читання й запис комірок EEPROM виконується через регістри введення/виведення EEAR (регістр адреси), EEDR (регістр даних) і EECR (регістр керування).

Регістр стану

Регістр стану - SREG є частиною простору введення/виведення й розташований за адресою $3F. У ньому встановлюються ознаки результату арифметичних операцій. Окремі біти регістра мають наступне призначення (рис. 1.4)

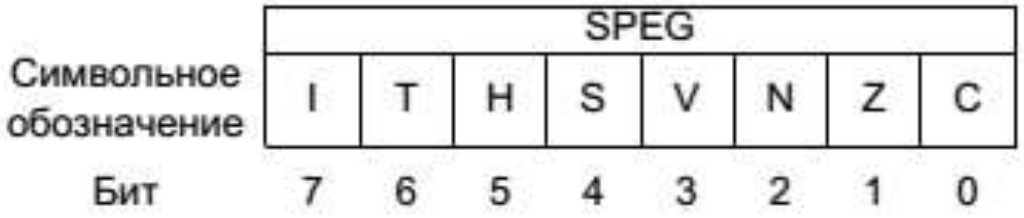


Рис. 1.4. Регістр стану SREG (Status Register)

Розглянемо деякі з ознак.

/Біт 0 - 3 (carry) - прапор переносу. Встановлюється, якщо під час виконання операції був перенос зі старшого розряду результату.

/Біт 1 - Z: (zero) - прапор нульового результату. Встановлюється, якщо результат операції дорівнює 0.

/Біт 2 - N - прапор негативного результату. Встановлюється, якщо MSB (Most Significant Bit - старший біт) результату дорівнює 1 (правильно показує знак результату, якщо не було переповнення розрядної сітки знакового числа).

/Біт 3 - V - прапор переповнення доповнення до двох. Встановлюється, якщо під час виконання операції було переповнення розрядної сітки знакового результату.

/Біт 4 - S - біт знака: S = N XOR V. Біт S завжди дорівнює виключаючому АБО між прапорами N (негативний результат) і V (переповнення доповнення до двох). Правильно показує знак результату й при переповненні розрядної сітки знакового числа.

/Біт 5 - H - прапор половинного переносу. Встановлюється, якщо під час виконання операції був перенос із 3-го розряду результату.

/Біт 6 - T - зберігання копіюємого біта. Команди копіювання битів BLD (Bit Loa) і BST (Bit STore) використають цей біт як джерело й приймач оброблюваного біта. Біт з регістра Регістрового файлу може бути скопійований в T командою BST, біт T може бути скопійований у біт Регістрового файлу командою BLD.

/Біт 7 - I - загальний дозвіл переривань. Для дозволу переривань цей біт повинен бути встановлений в одиницю. Керування перериваннями виробляється регістром маски переривань - GIMSK/TIMSK. Якщо прапор скинутий (0), незалежно від стану GIMSK/TIMSK, то переривання не дозволені.

Біт I очищається апаратно після входу в переривання й відновлюється командою RETI для дозволу обробки наступних переривань.

Переривання

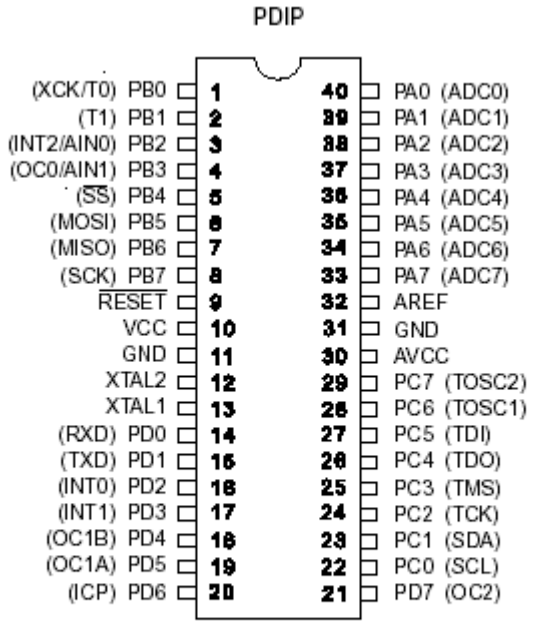
Описати, що відбувається усередині процесора під час виконання переривання, дуже просто. Якщо відбувся запит переривання, а прапор I у регістрі стану встановлений в 1, то адреса наступної команди зберігається в стеці, а виконання програми триває з адреси, що зберігається у відповідному векторі переривання. Коли запит переривання отриманий і програма перейшла по цьому вектору (адресі), то прапор I скидається в 0, щоб запобігти можливості виклику нового переривання під час обробки поточного переривання.

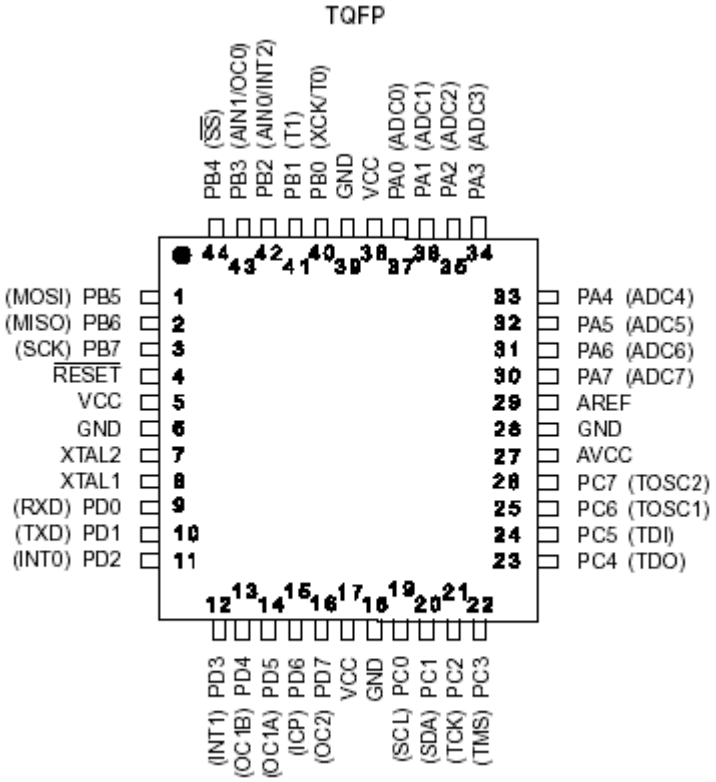
Прапор I буде знову встановлений в 1 наприкінці оброблювача переривання, коли виконується команда повернення RETI. Він також може бути встановлений в 1 у процесі обробки (після збереження контекстових регістрів), щоб дозволити вкладені переривання. Середні й старші моделі AVR можуть обробляти стільки вкладених переривань, на скільки вистачить обсягу стека для зберігання вмісту лічильника команд і регістрів контексту. Молодші моделі мають обмежений обсяг стека (три позиції), що може швидко переповнитися при виконанні вкладених переривань або підпрограм [1, 2, 3].

Загальна характеристика Atmega16 і робота с портами.

8-розрядний високопродуктивний AVR мікроконтролер з малим споживаннямПрогресивна RISC архітектура 130 високопродуктивних команд, більшість команд виконується за один тактовий цикл 32 8-розрядних робочих регістра загального призначення Повністю статична робота Наближається до Продуктивність 16 MIPS (при тактовій частоті 16 МГц) Вбудований 2-циклової перемножувачНезалежна пам'ять програм і даних 16 Кбайт внутрісистемного програмованої флеш-пам'яті (In-System Самопрограмірованний Flash) 1000 Забезпечує циклів стирання / запису Додатковий сектор завантажувальних кодів з незалежними битами блокування Внутрисистемное програмування вбудованої програмою завантаження Забезпечено режим одночасного читання / запису (Read-While-Write) 512 байт EEPROM Забезпечує 100000 циклів стирання / запису 1 кбайт SRAM вбудованої Програмована блокування, що забезпечує захист програмних засобів користувачаІнтерфейс JTAG (сумісний з IEEE 1149.1) Можливість сканування периферії, що відповідає стандарту JTAG Розширена підтримка вбудованої налагодження Програмування через JTAG інтерфейс: флеш, EEPROM пам'яті, перемичок і бітів блокуваннявбудована периферія Два 8-розрядних таймера / лічильника з окремим попередніми дільником, один з режимом порівняння 16-Один розрядний таймер / лічильник з окремим попередніми дільником і режимами захоплення і порівняння Лічильник реального часу з окремим генератором ЧОТИРИ PWM Каналу 8-канальний 10-розрядний аналого-цифровий перетворювач 8 несиметричних каналів 7 диференціальних каналів (тільки в корпусі TQFP) 2 диференціальних каналу з програмованим посиленням в 1, 10 або 200 разів (тільки в корпусі TQFP) -Байт 2-орієнтований провідний послідовний інтерфейс Програмований послідовний USART Послідовний інтерфейс SPI (провідний / ведений) Програмований

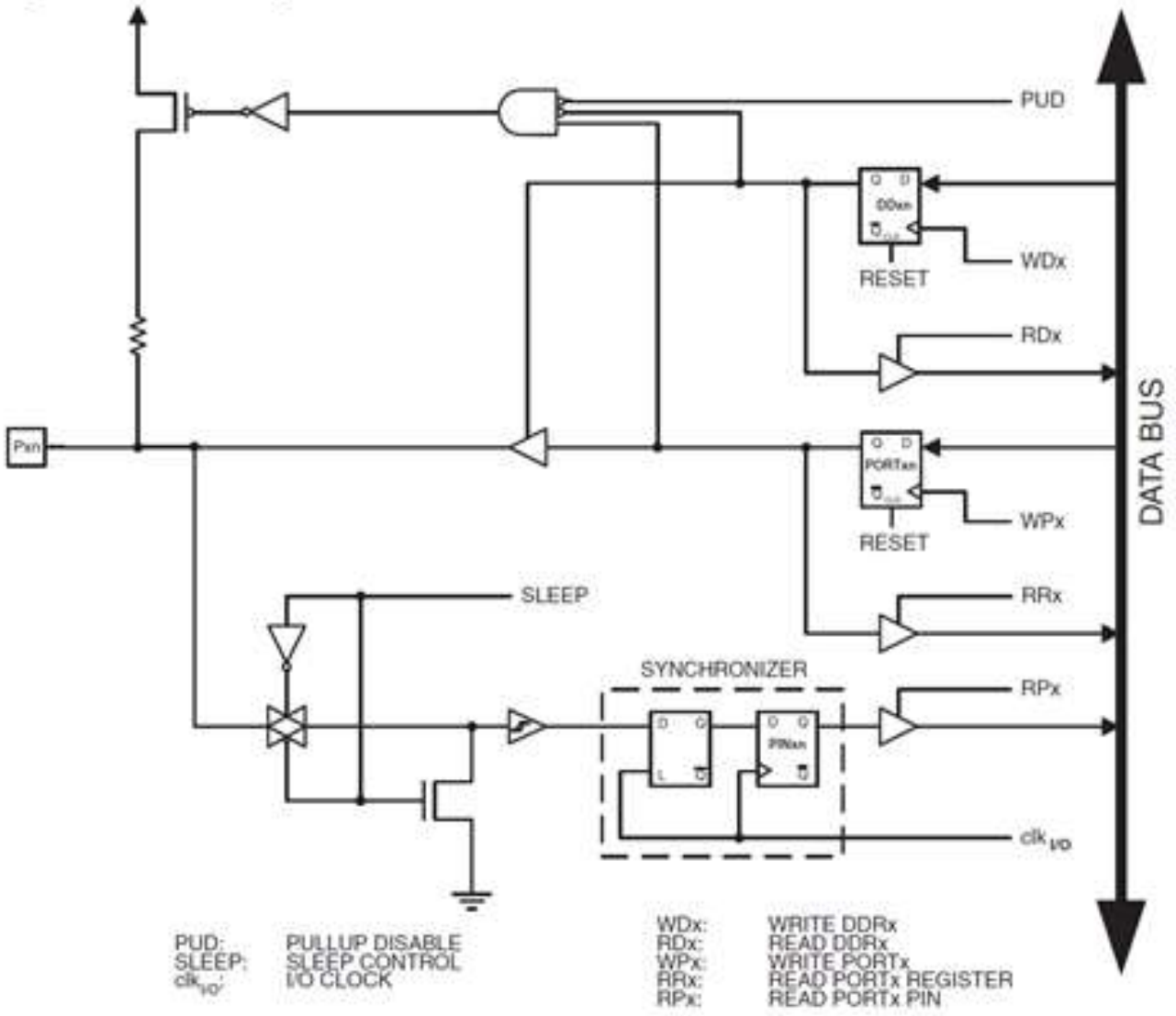
сторожовий таймер з окремим вбудованим генератором Вбудований аналоговий компараторСпеціальні мікроконтролерні функції За Скидання подачі живлення і програмований детектор короткочасного зниження напруги харчування Вбудований калібрований RC-генератор Внутрішні і зовнішні джерела переривань Шість режимів зниженого споживання: Idle, Power-зберегти, при відключенні харчування, режиму очікування Extended Standby і зниження шумів АЦПВисновки I / O і корпусу 32 програмовані лінії введення / виводу 40-вивідний корпус PDIP і 44-вивідний корпус TQFPробочі напруги 2,7 - 5,5 В (ATmega16L) 4,5 - 5,5 В (ATmega16)робоча частота 0 - 8 МГц (ATmega16L) 0 - 16 МГц (ATmega16)

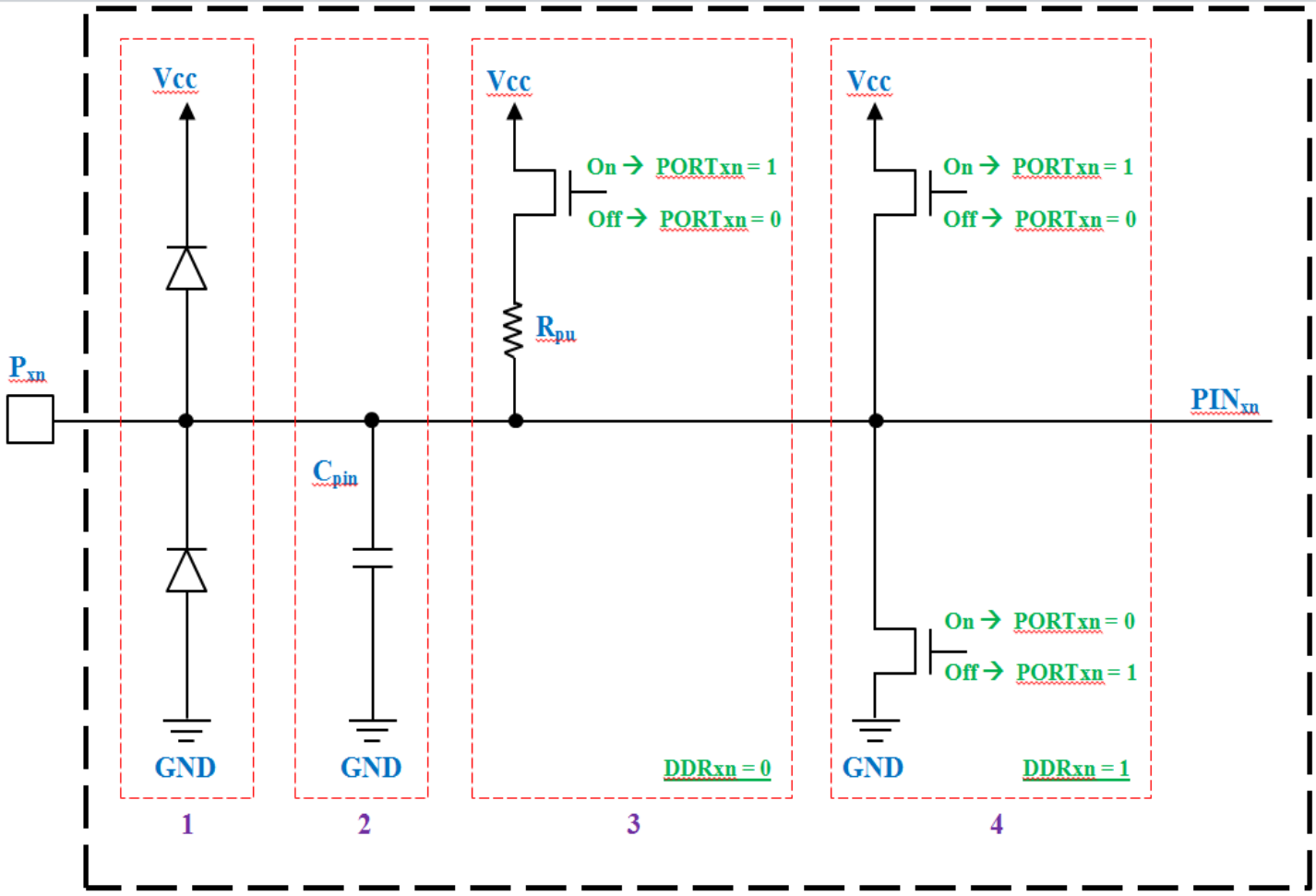




Із зовнішнім світом мікроконтролер спілкується через порти введення

виведення. Схема порту введення виведення вказана в даташіте: Але новачкові розібратися досить зі схемою досить складно. Тому схему спростимо:**Pxn** - ім'я ніжки порту мікроконтролера, де x буква порту (A, B, C або D), n номер розряду порту (7 ... 0). **Cpin** - паразитна ємність порту. **VCC** - напруга живлення. **Rpu** - відключається навантажувальний верхній резистор (pull-up).**PORTxn** - біт n регістра PORTx. **PINxn** - біт n регістра PINx. **DDRxn** - біт n регістра DDRx.





Розглянемо, що ж являє собою висновок мікроконтролера. На вході мікроконтролера стоїть невелика захист з двох діодів ***(см.1)*** , вона призначена для захисту введення мікроконтролера від короткочасних імпульсів напруги, що перевищують напруга живлення. Якщо напруга буде вище живлення, то верхній діод відкриється і це напруга буде стравлено на шину живлення, де з ним буде вже боротися джерело живлення і його фільтри. Якщо на введення потрапить негативне (нижче нульового рівня) напруга, то воно буде нейтралізовано через нижній діод і погаситься на землю. Втім, діоди там кволі і захист ця допомагає тільки від мікроскопічних імпульсів і перешкод. Якщо ж на ніжку мікроконтролера подати вольт 6-7 при 5 вольтах харчування, то внутрішні діоди його не врятують.

Конденсатор ***(см.2)*** - це паразитна ємність виведення. Хоч вона і крихітна, але присутній. Зазвичай її не враховують, але вона є.

Далі йдуть ключі управління***(см.3,4)*** . Кожен ключ підпорядкований логічному умові, які намальовані на малюнку. Коли умова виконується - ключ замикається. Кожен порт мікроконтролера AVR (зазвичай мають імена A, B і іноді C або навіть D) має 8 розрядів, кожен з яких прив'язаний до певної ніжці корпусу.

Кожен порт має три спеціальні регістра **DDRx** , **PORTx** і **PINx** (де x відповідає букві порту A, B, C або D). Призначення регістрів:

DDRx - Налаштування розрядів порту x на вхід або вихід.

PORTx - Управління станом виходів порту x (якщо відповідний розряд налаштований як вихід), або підключенням внутрішнього pull-up резистора (якщо відповідний розряд налаштований як вхід).

PINx -Читання логічних рівнів розрядів порту x. **PINxn** - це регістр читання. З

нього можна тільки читати. У регістрі **PINxn** міститься інформація про реальний поточному логічному рівні на висновках порту. Незалежно від налаштувань порту. Так що якщо хочемо дізнатися що у нас на вході - читаємо відповідний біт регістра**PINxn** .

Причому існує два кордони: межа гарантованого нуля і межа гарантованої одиниці - пороги за якими ми можемо однозначно чітко визначити поточний логічний рівень. Для пятівольтового харчування це 1.4 і 1.8 вольт відповідно. Тобто при зниженні напруги від максимуму до мінімуму біт в регістрі **PINx** переключиться з 1 на 0 тільки при зниженні напруга нижче 1.4 вольт, а ось коли напруга наростає від мінімуму до максимуму перемикання біта з 0 на 1 буде тільки після досягнення напруги в 1.8 вольта. Тобто виникає гістерезис перемикання з 0 на 1, що виключає хаотичні перемикання під впливом перешкод і наведень, а також виключає помилкове зчитування логічного рівня між порогами перемикання. При зниженні напруги живлення зрозуміло ці пороги також знижуються.

DDRxn - це регістр напрямку порту.Порт в конкретний момент часу може бути або входом або виходом (але для стану бітів PINxn це значення не має. Читати з **PINxn** реальне значення можна завжди).

DDRxy = 0 - висновок працює як ВХІД. *DDRxy* = 1 - висновок працює на ВИХІД.

PORTxn - режим управління станом виведення. Коли ми налаштовуємо висновок на вхід, то від **PORTx** залежить тип входу.

Коли ніжка налаштована на вихід, то значення відповідного біта в регістрі **PORTx** визначає стан виведення. Якщо **PORTxn =** 1 то на виведення лог.1, якщо **PORTxn =** 0 то на виведення лог.0. Коли ніжка налаштована на вхід, то якщо **PORTxn =** 0, то висновок в режимі Hi-Z. Якщо**PORTxn =** 1 то висновок в режимі PullUpc підтяжкою резистором в 100к до харчування. Таблиця. Конфігурація висновків портів. DDRxn PORTxn I / O Comment 0 0 I (Input) Вхід Високоімпендансний вхід. (Не рекомендую використовувати, так як можуть наводиться наведення від харчування) 0 1 I (Input) Вхід підтягти внутрішньо опір. 1 0 O (Output) Вихід На виході низький рівень. 1 1 O (Output) Вихід На виході високий рівень.

Загальна картина роботи порту показана на малюнках:

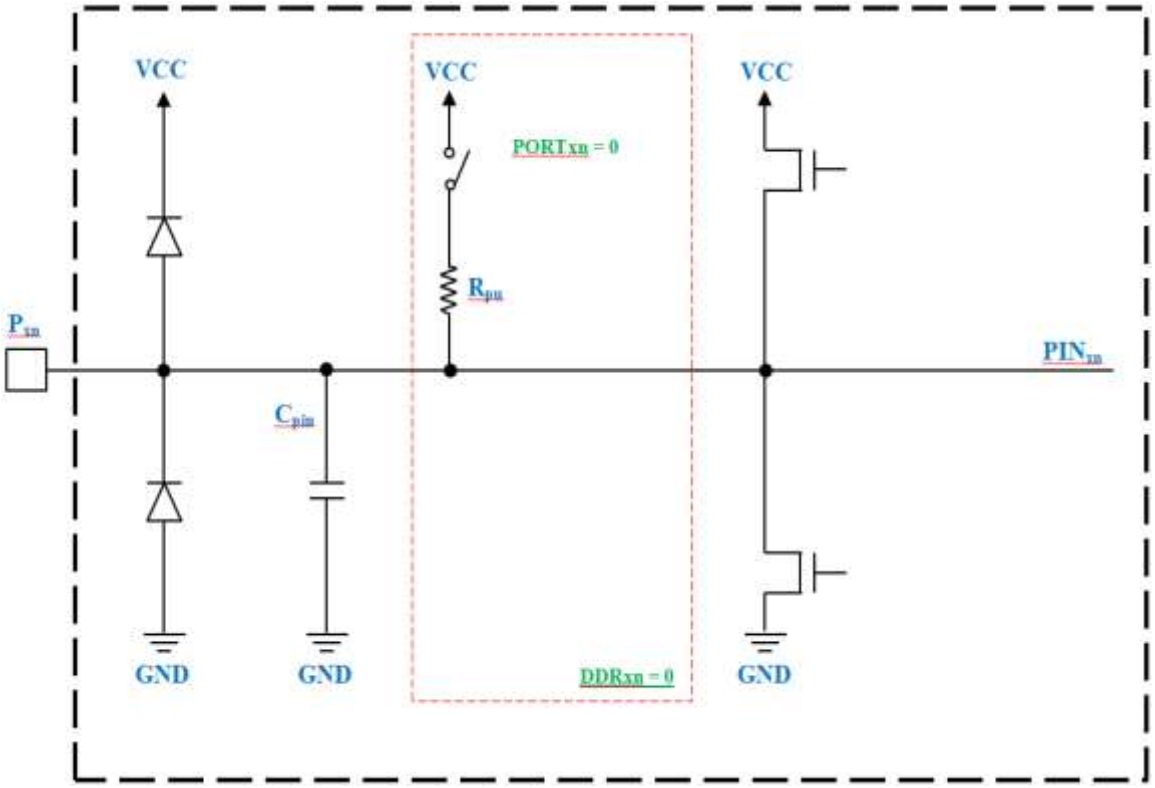


Рис. **DDRxn** = 0 **PORTxn** = 0 - Режим:**HI-Z** - високо імпендансний вхід.

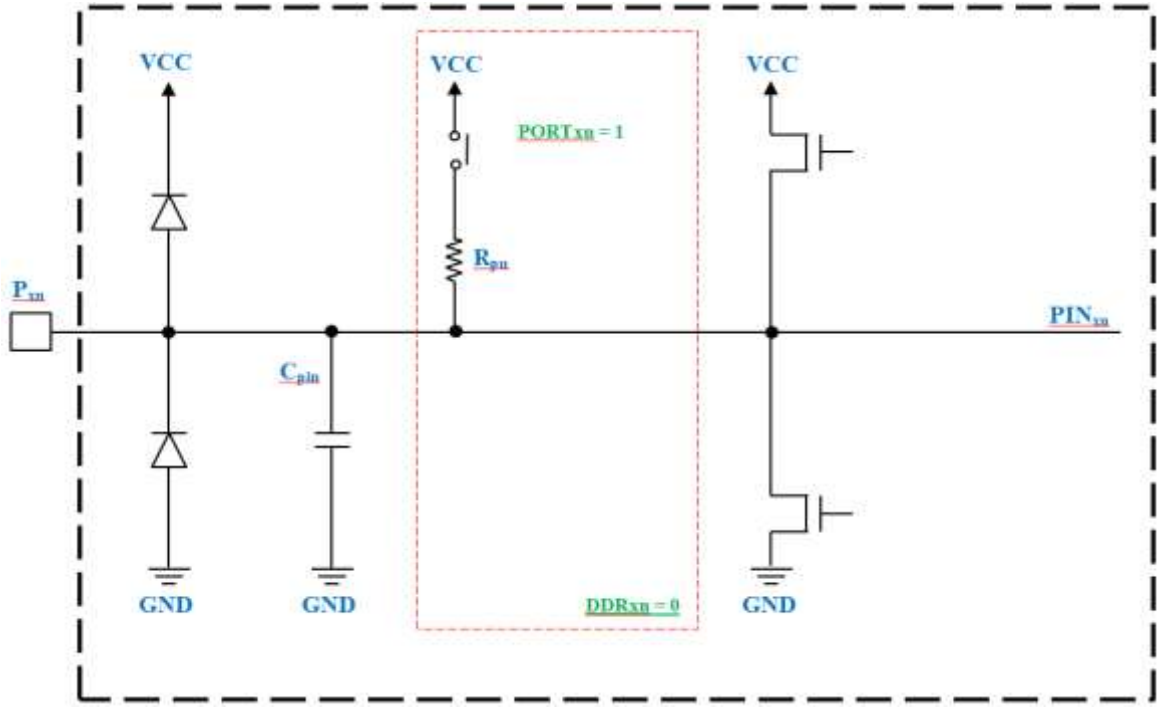


Рис. **DDRxn** = 0 **PORTxn** = 1 - Режим:**PullUp** - вхід з підтяжкою до лог.1.

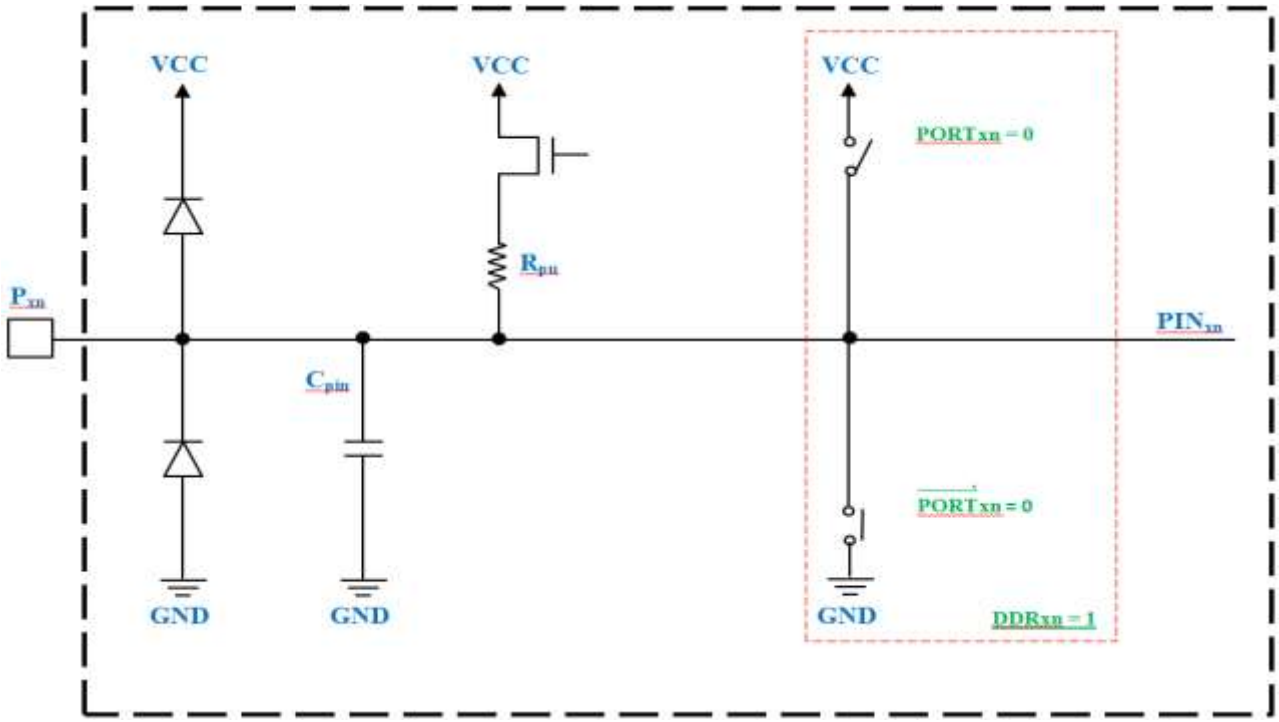


Рис. **DDRxn** = 1 **PORTxn** = 0 - Режим:**Вихід** - на виході лог.0. (майже GND)

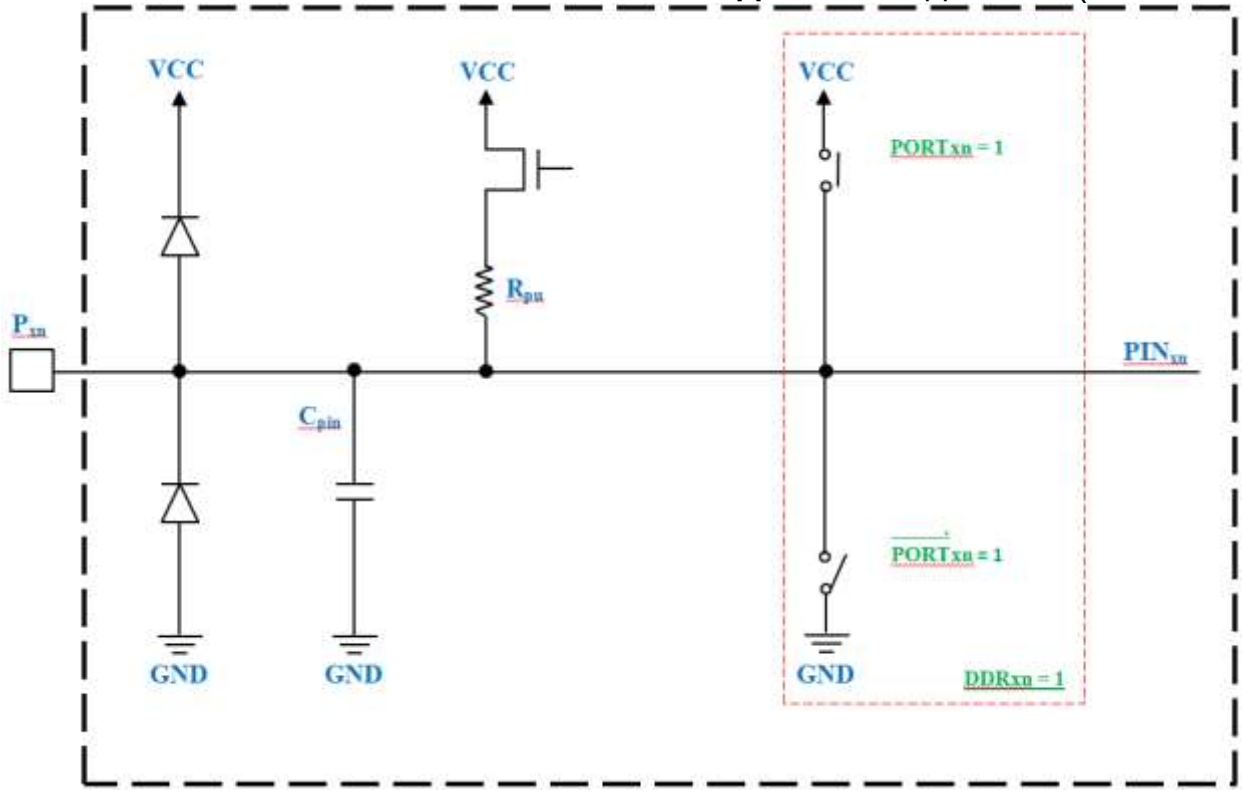


Рис. **DDRxn** = 1 **PORTxn** = 1 - Режим:**Вихід** - на виході лог.1. (майже VCC)

*Вхід **Hi-Z** - режим високоімпендансного входу.* Цей режим включений за замовчуванням. Всі ключі розімкнуті, а опір порту дуже велике. В принципі, в порівнянні з іншими режимами, можна його вважати нескінченністю. Тобто електрично висновок як би взагалі нікуди не підключений і ні на що не впливає. При цьому він постійно зчитує свої статки в регістр **PINn** і ми завжди можемо дізнатися що у нас на вході - одиниця або нуль. Цей режим хороший для прослуховування будь-якої шини даних, тому що він не робить на шину

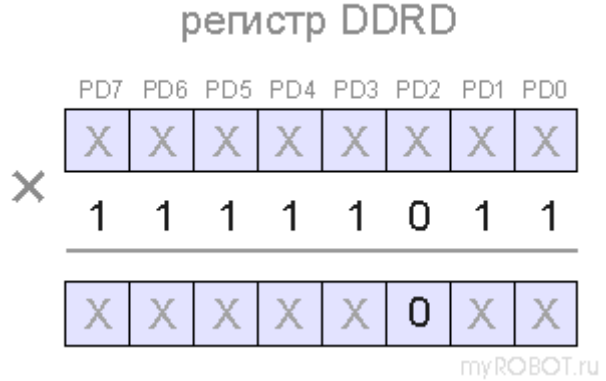
ніякого впливу. А що буде якщо вхід висить в повітрі? А в цьому випадку напруга буде на ньому скакати в залежності від зовнішніх наведень, електромагнітних. Дуже часто на порту в цьому випадку нестабільний синус 50Гц - наводка від мережі 220В, а в регістрі **PINn** буде змінюватися 0 і 1 з частотою близько 50 Гц. *Вхід**PullUp** - вхід з підтяжкою.* При **DDRxn** = 0 і **PORTxn =** 1 замикається ключ підтяжки і до лінії підключається резистор в 100кОм, що моментально призводить не підключену нікуди лінію в стан лог.1. Мета підтяжки очевидна - не допустити хаотичного зміни стану на вході під дією наведень. Але якщо на вході з'явиться логічний нуль (замикання лінії на землю кнопкою або іншим мікро контролером / мікросхемою), то слабкий 100кОмний резистор не зможе утримувати напругу на лінії на рівні лог.1 і на вході буде лог.0. *Режим виходу.* Якщо нам треба видати в порт лог.1, ми включаємо порт на вихід (**DDRxn =** 1) і видаємо лог.1 (**PORTxn** = 1) - при цьому замикається верхній ключ і на виведення з'являється напруга, близьке до харчування. А якщо треба лог.0, то включаємо порт на вихід (**DDRxn =** 1) і видаємо лог.0 (**PORTxn** = 1) - при цьому відкривається вже нижній вентиль, що дає на виведення близько нуля вольт.

Порти мікроконтролера - це пристрої введення / виводу, що дозволяють микроконтроллеру передавати або приймати дані. Стандартний порт мікроконтролера AVR має вісім розрядів даних, які можуть передаватися або прийматися паралельно. Кожному розряду (або біту) відповідає висновок (ніжка) мікроконтролера. Ніжки мікроконтролера також називають пинами. Для позначення портів використовуються латинські літери A, B, C і т.д. Кількість портів введення / виводу варіюється в залежності від моделі мікроконтролера. Будь порт мікроконтролера можна конфігурувати як вхід або як вихід. Для того щоб це зробити, слід записати в відповідний порту регістр **DDRx** необхідне значення. Крім того, як вхід або вихід можна конфігурувати окремо будь висновок (пін) порту. У будь-якому випадку, хочете ви конфігурувати весь порт або окремий висновок, вам необхідно буде працювати з рігістрамі DDRx. DDRx - регістр напрямку передачі даних. Цей регістр визначає, є той чи інший висновок порту входом або виходом. Якщо деякий розряд регістра DDRx містить логічну одиницю, то відповідний висновок порту налаштований як вихід, в іншому випадку - як вхід. Буква x в даному випадку повинна позначати ім'я порту, з яким ви працюєте. Таким чином, для порту A це буде регістр DDRA, для порту B - регістр DDRB і т. Д. Використовуючи **AVR GCC** , записати в необхідний регістр ту чи іншу значення можна одним з таких способів. Для всього порту відразу.**DDRD = 0xff;** Все висновки порту D будуть налаштовані як виходи. **0xff** - шестнадцатиричное уявлення числа ff, де 0x є префіксом, використовуваним для запису шістнадцяткових чисел. У десятковому поданні це буде число 255, а в двійковому вигляді воно буде виглядати як 11111111. Тобто у всіх бітах регістра DDRD будуть записані логічні одиниці. У **AVR GCC** для подання двійкових чисел використовується префікс 0b. Таким чином, число 11111111 має представлятися в програмі як 0b11111111. Ми можемо записати попередню команду в більш читабельному вигляді.**DDRD = 0b11111111;** Хоча такий запис і виглядає більш наочною, при конфігурації портів прийнято використовувати шестнадцатиричное уявлення чисел. Для того щоб настроїти всі висновки порту D як входи, слід записати в усі біти регістра DDRD логічні нулі. **DDRD = 0x00;** В регістр DDRD можна записати і інші числа. Наприклад: **DDRD = 0xb3; 0xb3** - шестнадцатиричное уявлення числа 179. У двійковому вигляді воно буде виглядати

як 10110011. Тобто частина висновків порту D буде налаштована як виходи, а частина - як входи. **PD0 - 1 (вихід) PD1 - 1 (вихід) PD2 - 0 (вхід) PD3 - 0 (вхід) PD4 - 1 (вихід) PD5 - 1 (вихід) PD6 - 0 (вхід) PD7 - 1 (вихід)**

Кожен біт регістрів DDRx може бути встановлений окремо. Наприклад, щоб настроїти окремо висновок PD2 як вихід, нам необхідно в відповідний біт регістра DDRD записати 1. Для цього застосовують наступну конструкцію. **DDRD | = 1 << 2; 1 << 2** - здійснює зрушення одиничкі вліво на 2 біти, тобто справа додаються два нульових біта і виходить 100, а знак "|", що стоїть перед знаком присвоювання "=", здійснює операцію побітного логічного складання. При логічному складення **0 + 0 = 0, 0 + 1 = 1, 1 + 1 = 1** . Операцію логічного додавання по-іншому називають операцією **АБО**(англійська назва OR). Таким чином, до бітам, що зберігаються в регістрі DDRD, додається двоичное 100, представлене в 8-бітному регістрі мікроконтролера як 00000100, і результат записується назад в регістр DDRD.

Щоб настроїти окремо висновок PD2 як вхід, нам необхідно в відповідний біт регістра DDRD записати 0. Для цього застосовують наступну конструкцію. **DDRD & = ~ (1 << 2);** В даному випадку результат зсуву одиниці на дві позиції вліво інвертується за допомогою операції побітного инвертирования, що позначається значком " ~ ". При інверсії ми отримуємо замість нулів одиниці, а замість одиничок - нулі. Ця логічна операція інакше називається операцією **НЕ** (англійська назва NOT). Таким чином, при побітному інвертуванні 00000100 ми отримуємо 11111011. Число, що за допомогою операції побітного логічного множення & множиться на число, яке зберігається в регістрі DDRD, і результат записується в регістр DDRD. При логічному множенні **0 * 0 = 0, 0 * 1 = 0, 1 * 1 = 1** . Операцію логічного множення інакше називають операцією **І** (англійська назва AND). Тобто зрушена нами вліво на дві позиції одиничка перетворюється при інвертуванні в нуль і множиться на відповідний біт, що зберігається в регістрі DDRD. При множенні на нуль ми отримуємо нуль. Таким чином, біт PD2 стає рівним нулю.

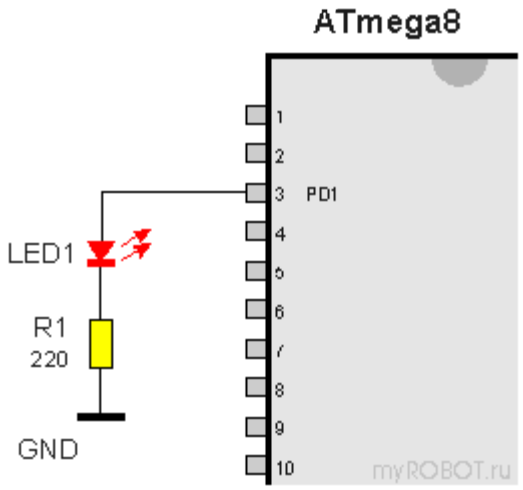


Крім логічних операцій І, АБО, НЕ існує також операція "виключає АБО" (англійська назва XOR). Вона позначається значком ^. При виключає АБО значення біта, до якого "додається" одиничка, змінюється на протилежне.Наприклад, 110011 ^ 11010 = 101001.

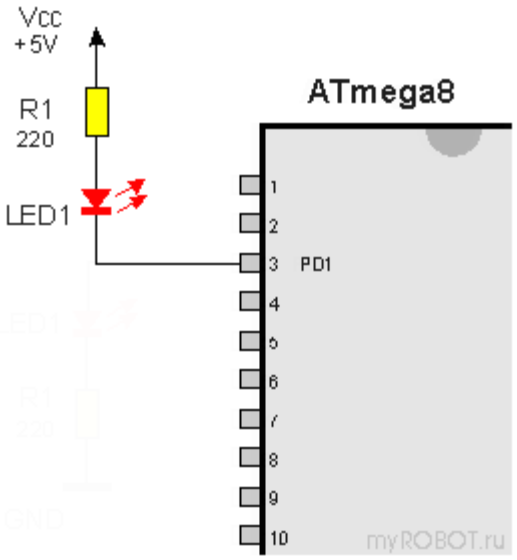
Слід додати, що робота з числами в 8-бітному мікроконтролері проходить з використанням 8-бітних регістрів. Перед обчисленнями аргумент поміщається в один із спеціальних регістрів, з якими безпосередньо може працювати арифметико-логічний пристрій (АЛП).Наприклад, перед виконанням команди **DDRD & = ~ (1 << 2)** аргумент поміщається в допоміжний регістр мікроконтролера. Вміст такого регістра буде виглядати як **11111011** .
Після цього здійснюється операція побітного множення, що дає в другому бите регістру DDRD значення **0** .

Після того як напрямок передачі даних у порту налаштоване, можна привласнити порту значення, яке буде зберігатися у відповідному регістрі **PORTx** .
PORTx - регістр порту, де x позначає ім'я порту. Якщо висновок сконфигурирован як, то одиничка у відповідному бите регістру PORTx формує на виведення сигнал високого рівня, а нуль - сигнал низького рівня. Якщо ж висновок налаштований як вхід, то одиничка у відповідному бите регістру PORTx підключає до висновку внутрішній підтягаючий pull-up резистор, який забезпечує високий рівень на вході при відсутності зовнішнього сигналу. Встановити "1" на всіх висновках порту D можна наступним чином. **PORTD = 0xff;** А встановити "0" на всіх висновках порту D можна так.**PORTD = 0x00;** до кожного біту регістрів PORTx можна звертатися і окремо так само, як у випадку з регістрами DDRx. наприклад, команда **PORTD | = 1 << 3;** встановить "1" (сигнал високого рівня) на виведення PD3. команда **PORTD & = ~ (1 << 4);** встановить "0" (сигнал низького рівня) на виведення PD4. В **AVR GCC** зрушення можна здійснювати і за допомогою функції **_BV ()** , яка виконує порозрядному зрушення і вставляє результат в компільований код. у разі використання функції _BV () дві попередні команди будуть виглядати наступним чином.
PORTD | = _BV (PD3); // встановити "1" на лінії 3 порту D
PORTD & = ~ _BV (PD4); // встановити "0" на лінії 4 порту D

Тепер спробуємо написати кілька простих програм для кращого розуміння принципу роботи з портами мікроконтролера. Перші наші програми будуть складатися всього з декількох рядків, і в їх завдання входитиме запалювання світлодіода, підключеного до мікроконтролеру. Підключити світлодіод до мікроконтролеру можна різними способами.



Малюнок 1



малюнок 2

Залежно від способу підключення світлодіод буде загорятися або від сигналу високого рівня, що подається на висновок PD1 мікроконтролера, як в першому випадку, або від сигналу низького рівня в разі підключення, зображеного на другому малюнку.

```
/ ********************************************** ************************
ПРИКЛАД включенні світлодіод СИГНАЛОМ ВИСОКОГО РІВНЯ
Приклад підключення на малюнку 1
********************************************** ************************ /

  #include <avr / io.h>
  int main ( void ) {   // початок основної програми
  DDRD = 0xff;                  // все висновки порту D конфігурувати як виходи
  PORTD | = _BV (PD1);          // встановити "1" (високий рівень ) на виведення PD1
  }                             // закриває дужка основної програми
```

```
/ ********************************************** ********************
ПРИКЛАД включенні світлодіод СИГНАЛОМ НИЗЬКОЇ РІВНЯ
Приклад підключення на малюнку 2
********************************************** ********************* /

  #include <avr / io.h>

  int main ( void ) {   // початок основної програми
  DDRD = 0xff;                  // все висновки порту D конфігурувати як виходи
  PORTD & = ~ _BV (PD1);        // встановити "0" (низький рівень) на виведення PD1
  }                             // закриває дужка основної програми
```

Тепер спробуємо моргнути світлодіодом, підключеним так, як це зображено на лівому малюнку. Для цього використовуємо функцію затримки _delay_ms (). Функція _delay_ms () формує затримку в залежності від переданого їй аргументу, вираженого в мілісекундах (в одній секунді 1000 мілісекунд). Максимальна затримка може досягати 262.14 мілісекунд. Якщо користувач передасть функції значення більш 262.14, то відбудеться автоматичне зменшення дозволу до 1/10 мілісекунди, що забезпечує затримки до 6.5535 секунд. Функція _delay_ms () міститься в файлі delay.h, тому нам буде необхідно підключити цей файл до програми. Крім того, для нормальної роботи цієї функції необхідно вказати частоту, на якій працює мікроконтролер, в герцах.

```
/ ************************************
ПРИКЛАД мигання світлодіода
Приклад підключення на малюнку 1
************************************* /

  #define F_CPU 1000000UL   // вказуємо частоту в герцах

  #include <avr / io.h>
  #include <util / delay.h>

  int main ( void ) {   // початок основної програми

  DDRD = 0xff;                    // все висновки порту D конфігурувати як виходи

  PORTD | = _BV (PD1);          // встановити "1" (високий рівень) на виведення PD1,
                                // Запалити світлодіод

  _delay_ms (500);   // чекаємо 0.5 сек.

  PORTD & = ~ _BV (PD1);      // встановити "0" (низький рівень) на виведення PD1,
                                // Погасити світлодіод

  _delay_ms (500);   // чекаємо 0.5 сек.

  PORTD | = _BV (PD1);          // встановити "1" (високий рівень) на виведення PD1,
                                // Запалити світлодіод

  _delay_ms (500);   // чекаємо 0.5 сек.

  PORTD & = ~ _BV (PD1);      // встановити "0" (низький рівень) на виведення PD1,
                                // Погасити світлодіод

  }                               // Закриває дужка основної програми
```

Серія спалахів світлодіодом буде дуже короткою. Для того щоб зробити миготіння

безперервним, можна організувати нескінченний цикл за допомогою оператора безумовного переходу "goto".Оператор goto здійснює перехід до місця програми, позначеному міткою. Ім'я мітки не повинно містити пропусків. Після імені мітки ставиться знак двокрапки. Між ім'ям мітки і двокрапкою не повинно бути пробілів.

```
/ ********************************************** ******
ПРИКЛАД нескінченно мигання світлодіода
Приклад підключення на малюнку 1
********************************************** ****** /

  #define F_CPU 1000000UL   // вказуємо частоту в герцах

  #include <avr / io.h>
  #include <util / delay.h>

  int main ( void ) {   // початок основної програми

  DDRD = 0xff;                    // все висновки порту D конфігурувати як виходи

start:                            // мітка для команди goto start

  PORTD | = _BV (PD1);            // встановити "1" (високий рівень) на виведення PD1,
                                  // Запалити світлодіод

  _delay_ms (250);   // чекаємо 0.25 сек.

  PORTD & = ~ _BV (PD1);          // встановити "0" (низький рівень) на виведення PD1,
                                  // Погасити світлодіод

  _delay_ms (250);   // чекаємо 0.25 сек.

goto start;                       // перейти до мітки start

  }                               // Закриває дужка основної програми
```

Лекція 3
Особливості програмування на мові Сі мікроконтролерів AVR.

1. Середовища для програмування на мові Сі.
2. Області пам’яті AVR та їх використання в мові Сі.
3. Особливості компіляції та структура програми на мові Сі.
4. Операнди й операції мови Сі.

1. Середовища для програмування на мові Сі.

Для мікроконтролерів AVR існують різні мови програмування, але, мабуть, найбільш придатними є асемблер і Сі, оскільки в цих мовах в найкращій мірі реалізовані всі необхідні можливості по управлінню апаратними засобами мікроконтролерів. Асемблер - це низькорівневий мову програмування, що використовує безпосередній набір інструкцій мікроконтролера. Створення програми на цій мові вимагає хорошого знання системи команд програмованого чіпа і достатнього часу на розробку програми. Асемблер програє Сі в швидкості і зручності розробки програм, але має помітні переваги в розмірі кінцевого виконуваного коду, а відповідно, і швидкості його виконання. Сі дозволяє створювати програми з набагато більшим комфортом, надаючи розробнику всі переваги мови високого рівня. Слід ще раз відзначити, що архітектура і система команд AVR створювалася за безпосередньої участі розробників компілятора мови Сі і в ній враховані особливості цієї мови.Компіляція вихідних текстів, написаних на Сі, здійснюється швидко і дає компактний, ефективний код. Основні переваги Сі перед ассемблером: висока швидкість розробки програм; універсальність, що не вимагає досконального вивчення архітектури мікроконтролера; найкраща документовані і читаність алгоритму; наявність бібліотек функцій; підтримка обчислень з плаваючою точкою. У мові Сі гармонійно поєднуються можливості програмування низького рівня з властивостями мови високого рівня. Можливість низкоуровневого програмування дозволяє легко оперувати безпосередньо апаратними засобами, а властивості мови високого рівня дозволяють створювати легко читається і модифікується програмний код. Крім того, практично всі компілятори Сі мають можливість використовувати асемблерні вставки для написання критичних за часом виконання і займаним ресурсів ділянок програми. Одним словом, Сі - найбільш зручний мову як для початківців знайомитися з мікроконтролерами AVR, так і для серйозних розробників.

Компілятори

Щоб перетворити вихідний текст програми в файл прошивання мікроконтролеру, застосовують компілятори. Фірма Atmel поставляє потужний компілятор асемблера, який входить в середу розробки AVR Studio, що працює під Windows. Поряд з компілятором, середовище розробки містить відладчик і емулятор. AVR Studio абсолютно безкоштовна і доступна на сайті Atmel . В даний час представлено досить багато компіляторів Сі для AVR. Найпотужнішим з них вважається компілятор фірми IAR Systems . Саме її співробітники в середині 90-х років брали участь в розробці системи команд AVR. IAR C Compiler має широкі можливості по оптимізації коду і поставляється в складі інтегрованого середовища розробки IAR Embedded Workbench (EWB), що включає в себе також компілятор асемблера, линкер, менеджер проектів і бібліотек, а також відладчик. Ціна повної версії пакету становить 2.820 EUR. ФірмоюImage Craft випускається компілятор мови Сі, який отримав досить широку популярність. Image Craft C Compiler володіє непоганим рівнем оптимізації коду і досить низькою ціною (від $ 199 до $ 749 в залежності

від версії). Не меншу популярність завоював Code Vision AVR C Compiler , ціна повної версії цього компілятора невисока і складає 150 EUR. Компілятор поставляється разом з інтегрованим середовищем розробки, в яку, крім стандартних можливостей, включена досить цікава функція - CodeWizardAVR Automatic Program Generator. Наявність в середовищі розробки послідовного терміналу дозволяє виробляти налагодження програм з використанням послідовного порту мікроконтролера. Воістину культової стала інтегрована середовище розробки WinAVR .Вона включає потужні компілятори Сі і асемблера, програматор AVRDUDE, відладчик, симулятор і безліч інших допоміжних програм і утиліт. WinAVR прекрасно інтегрується з середовищем розробки AVR Studio від Atmel. Асемблер ідентичний по вхідному коду асемблеру AVR Studio. Компілятори Сі і асемблера мають можливість створення налагоджувальних файлів у форматі COFF, що дозволяє застосовувати не тільки вбудовані засоби, але і використовувати потужний симулятор AVR Studio.Ще одним важливим плюсом є те, що WinAVR поширюється вільно без обмежень (виробники підтримують GNU General Public License).

2. Області пам’яті AVR та їх використання в мові Сі.

Мікроконтролери AVR реалізовані за принципом гарвардської архітектури, де пам'ять програм і пам'ять даних, як і шини доступу до них, розділені. Такий поділ дозволяє, наприклад, одночасно зчитувати нову асемблерну інструкцію з пам'яті програм по одній шині і в той же момент записувати результат попередньої команди в пам'ять даних мікроконтролера (ОЗУ / УВВ / РОН) за іншою.Таке "*поділ праці*" називається конвеєром (pipeline) і дозволяє виконувати по одній команді за такт.

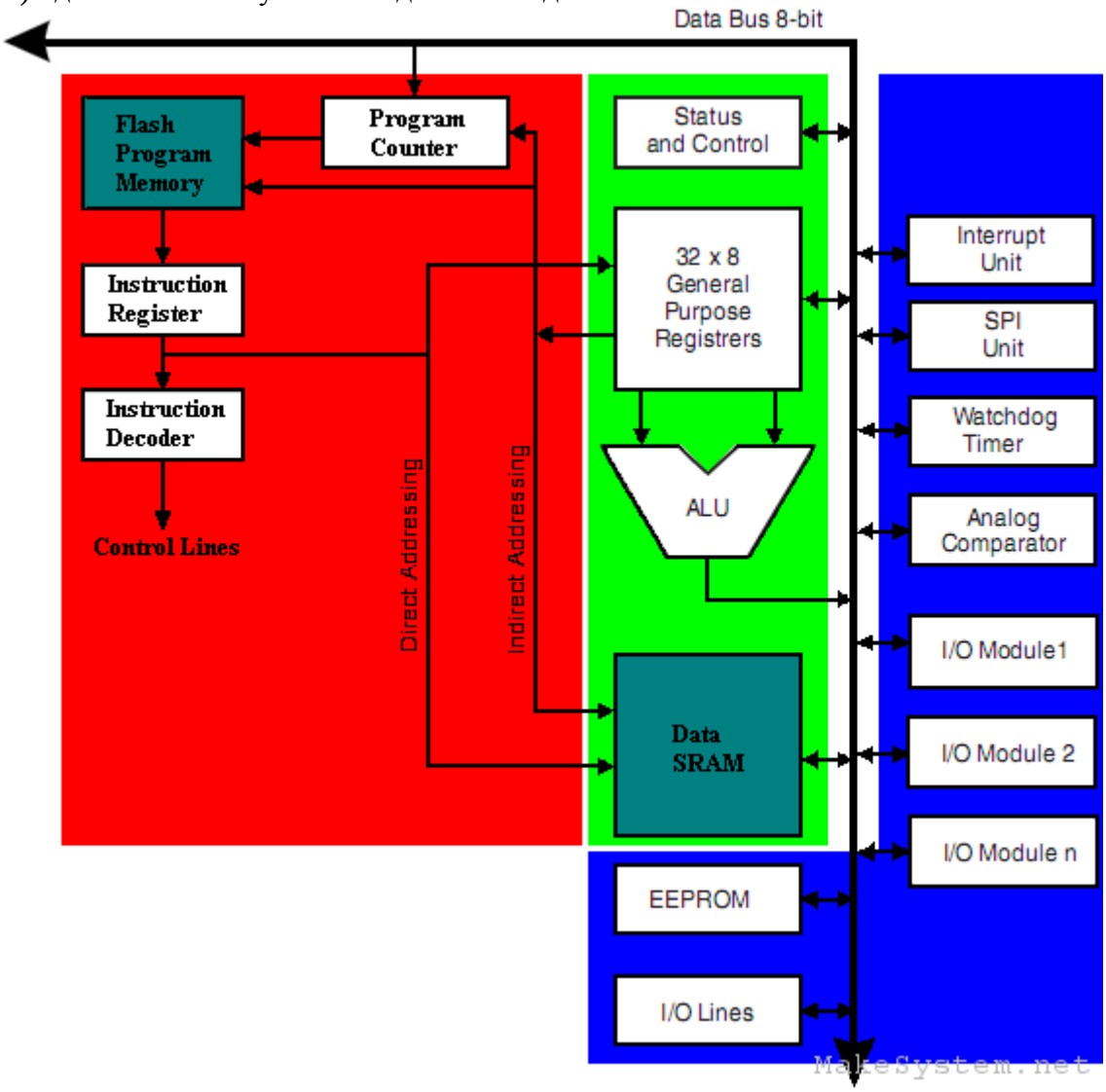


Рис.1 Архітектура AVR процесора

Пам'ять даних (Data Memory)

Одна з ключових складових AVR мікроконтролера є **8-бітна Шина Даних** (8-bit Data Bus), що зв'язує між собою ОЗУ, лічильник команд, регістр стану, регістри периферійних пристроїв а також RALU (Регістри загального призначення + АЛУ) мікроконтролера.

Пам'ять даних являє собою сукупність регістрів загального призначення (РОН), регістрів введення / виводу а також статичного ОЗУ (SRAM). У AVR є кілька наборів асемблерних команд які призначені спеціально для адресації, до регістрів введення / виводу (зі своїм адресним простором), а також набір команд для адресації через шину даних до всієї області пам'яті даних (зі своїм адресним простором).

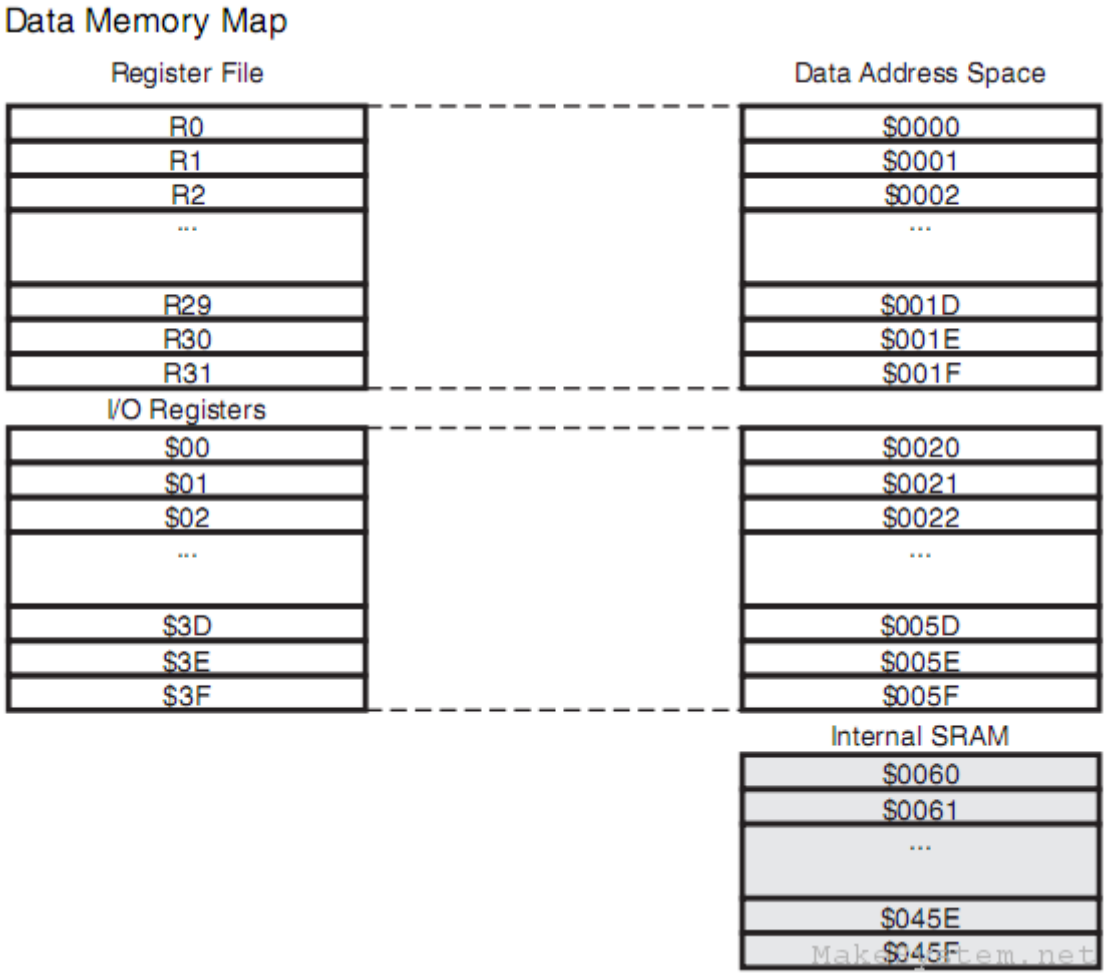


Рис.2 Пам'ять даних

При програмуванні на мові Сі, програміст може оперувати декількома типами даних:

char- 8-бітна змінна. **Int** , **short** - 16-бітові змінні. **Long** ,**float** , **double** - 32-бітові змінні.

А також **struct** , **union**

Якщо ми оголосимо змінну всередині функції, тобто зробимо її локальної, то компілятор виділить для неї один або кілька регістрів загального призначення. Якщо в даний момент не вистачає РОН, компілятор помістить змінну в оперативну пам'ять (точніше в Стек), що надалі вимагатиме додаткові такти для зчитування змінної з ОЗУ / Стека. Якщо ми оголосимо змінну за межами функції (зробимо її глабальной) або визначимо її як **static** , то компілятор не роздумуючи виділить для неї місце в ОЗУ пам'яті. Так що рада: намагайтеся, по можливості, оголошувати змінні всередині функцій, тим самим надавши компілятору можливість для вибору найбільш оптимального розміщення змінних.

Також в мові Сі можна *рекомендувати* (необов'язковий для виконання) компілятору помістити ту чи іншу змінну в РОН за допомогою ключового слова **register** .

```
register unsigned char value ; // може бути поміщена в РОН
static unsigned int data ; // буде поміщена в ОЗУ
```

Оскільки AVR, як і більшість 8-бітних мікроконтролерів, не вміє (апаратно) працювати з 16- / 32- / 64-бітних змінними, для таких випадків розроблені бібліотечні функції, що входять до складу компіляторів високорівневих мов, таких як Сі. Ці функції реалізують велику кількість різноманітних математичних / логічних і ін. Операцій, таких як множення / ділення 16- / 32- / 64-бітних змінних зі знаком і без, робота з числами з плаваючою комою, робота зі структурами, бітовими полями і т. д. Але як ви розумієте використання змінних даного типу, а значить і пов'язані з ними функції, призводить до збільшення числа тактів необхідних для читання / запису а також регістрів загального призначення використовуваних при адресації до таких змінних, а найголовніше, хоча і невелике, але додаткове місце в пам'яті програм, для зберігання цих функцій.

Додаткові модифікатори.

eeprom - розмістити змінну в EEPROM. Це незалежна пам'ять - значення таких змінних зберігається при виключенні живлення і при перезавантаженні МК.

приклад:

eeprom unsigned int x;

Якщо це перша змінна в EEPROM то її молодший байт буде поміщений в клітинку 1 EEPROM а старший в клітинку 2.

volatile - ставте якщо потрібно запобігти можливості пошкодження вмісту змінної в перериванні, і не дозволити компілятору спробувати викинути її при оптимізації коду. Ставте завжди якщо не знаєте точно - потрібно чи ні! приклад: volatile unsigned char x;

Пам'ять програм (Flash Program Memory)

Програма мікроконтролера по суті являє собою один великий масив, кожна осередок якого містить одну асемблерну інструкцію, в свою чергу одна асемблерна інструкція займає 2 байта пам'яті програм. Термін дії пам'яті програм в AVR мікроконтролери розрахований на 10000 циклів запису / стирання і може зберігати інформацію протягом 20 років при температурі в 85 ° C і до 100 років при температурі в 25 ° C.

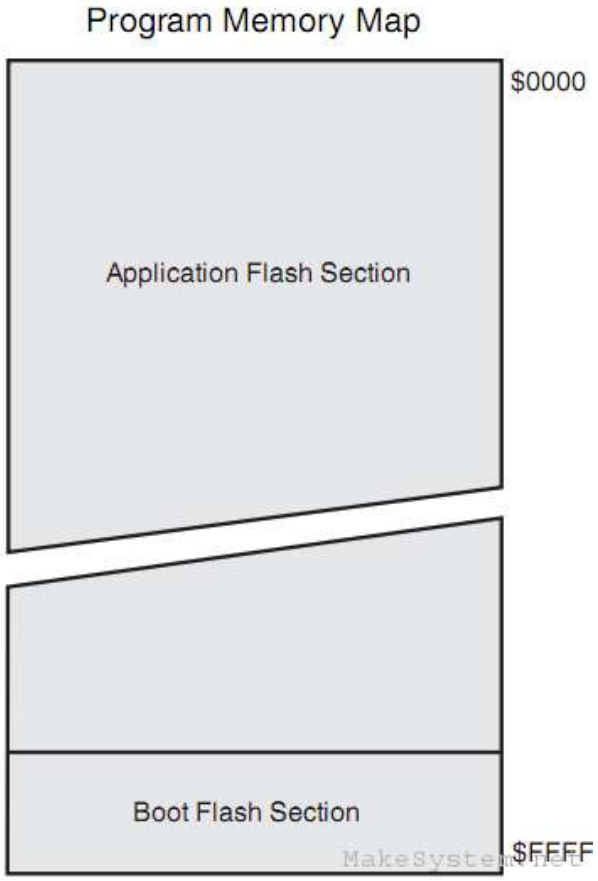


Рис.3 Пам'ять програм

Память EEPROM.

МК AVR від Atmel містять також EEPROM (Electrically Erasable Read-Only Memory) - незалежну пам'ять з досить великою кількістю циклів запису (для Atmega16 512 байт). Дані, записані в цю пам'ять, не будуть скидатися навіть при відключенні харчування, що дуже зручно, наприклад, для зберігання налаштувань або якихось ідентифікаційних даних. EEPROM в AVR має обмежену кількість циклів запису - 100 000. Кількість циклів читання не обмежена.

Доступ до EEPROM в AVR можна отримати за допомогою спеціальних регістрів, які відповідають за адреса, інформацію для запису (або прочитану інформацію) і здійснюється дію (запис / читання). У мові С немає будь-яких стандартів доступу або адресації пам'яті (. Тому кожен компілятор використовує свій метод зберігання інформації в пам'яті. Використання бібліотеки AVRLibC EEPROM AVRLibC (зазвичай вона входить до складу компілятора AVR-GCC) містить готову бібліотеку для роботи з EEPROM. Щоб її використовувати, потрібно додати наступний заголовки:#include <avr / eeprom.h>

У цій бібліотеці три основних типи даних: byte (1 байт), word (2 байта) і блок даних. У нових версіях додані ще два типи - dword (4 байта) і float (8 байт. Для кожного типу є своя функція запису і читання.

Читаємо \ пишемо по одному байту (byte)
uint8_t eeprom_read_byte (const uint8_t * addr)
void eeprom_write_byte (uint8_t * addr, uint8_t value)
// Читаємо \ пишемо по два байта (word)
uint16_t eeprom_read_word (const uint16_t * addr)
void eeprom_write_word (uint16_t * addr, uint16_t value)
// Читаємо \ пишемо блокамиvoid
eeprom_read_block (void * pointer_ram, const void * pointer_eeprom, size_t n)
void eeprom_write_block (const void * pointer_ram, void * pointer_eeprom size_t n);
Константи.

flash і const ставляться перед оголошенням констант - незмінних даних зберігаються у флеш пам'яті програм. Вони дозволяють вам використовувати не зайняту програмою пам'ять МК. Зазвичай для зберігання строкових даних - різні інформаційні повідомлення, або чисел і масивів чисел.

Константи flash int integer_constant = 1234 + 5;
flash char char_constant = 'a';
flash long long_int_constant1 = 99L;
flash long long_int_constant2 = 0x10000000;
flash int integer_array1 [] = {1,2,3};
flash int integer_array2 [10] = {1,2};
flash int multidim_array [2] [3] = {{1,2,3}, {4,5,6}};
flash char string_constant1 [] = "This is a string constant";
const char string_constant2 [] = "This is also a string constant";

Регістри процесора

Регістри , є. За допомогою регістрів і т.д. значення змінних, адреси переходів і т.п., через регістри виробляється настройка і управління периферійними пристроями. Основні регістри учавствующие в обчисленнях, контролі, налаштування і упраління як периферійними пристроями так і всього мікроконтролера є: регістровий файл загального призначення, регістр стану, лічильник команд, покажчик стека, регістри введення / виводу.

Регістровий Файл загального призначення (**General Purpose Register File**) - регістровий файл складається з 32 регістрів загального призначення. Ці регістри використовуються в якості операндів в процесі виконання команд. При програмуванні на С, програмісту надається можливість оперувати змінними, які в поледствіі будуть поміщені компілятором в регістровий файл (РОН) або ОЗУ мікроконтролера. Тобто якийсь рівень абстракції. При програмуванні на асемблері, програміст працює безпосередньо з регістрами загального призначення, адресами ОЗУ і т.д.

Регістр стану (**Status Register**) - даний реєстр містить інформацію про результат виконання попередньої ассемблерной інтсрукціі. За допомогою даного результату, можна

змінювати хід виконання (розгалуження) програми, тобто приймати рішення в залежності від результату попередньої операції. Арифметичні, логічні і операції з битами призводять до зміни регістра стану. Всі інші асемблерні команди не змінюють поля регістру состаяния.

Лічильник команд (**Program counter**) - в цьому регістрі міститься адреса однієї комірки пам'яті програм (2 байта), тобто адреса ассемблерной інструкції яку на наступному такті належить витягти і передати на виконання. При подачі напруги харчування (включенні мікроконтролера), гарантовано що в лічильнику команд, буде знаходитися значення 0 (нуль), що означає що наступна команда (яка буде передана на виконання) розташована за адресою 0 (нуль) пам'яті програм, тобто найперша команда .

У разі умовних або безумовних переходів, то в лічильник команд завантажується адреса тієї інструкції яка повинна бути виконана наступної.

Покажчик стека (**Stack pointer**) - Стек використовується в основному для зберігання **тимчасових даних** , таких як локальні змінні (ті самі що не поміщаються в регістри загального призначення), а також адреси повернень з підпрограм / переривань.Слід зазначити, що стек зазвичай розташований у верхній області ОЗУ, починаючи з самої останньої ОЗУ осередки (RAM_END). У нижній ОЗУ, починаючи з адреси 0 × 60 пам'яті даних, розташовані глобальні змінні, масиви даних, константи, область пам'яті звана Купа (Heap), яка використовується для динамічного розподілу пам'яті і т.д.

Покажчик стека є два 8-бітних регістра, які завжди вказують на вершину стека, тобто в них зберігається адреса останньої вміщеній в стек змінної. Читати / записувати дані можна тільки за адресою міститься в покажчику стека. У той час як " *нижня частина* " ОЗУ пам'яті залишається незмінною, стек росте назустріч даними зберігаються в ОЗП і може трапиться так, що дані стека перепишуть дані ОЗУ. Якщо така ситуація станеться то хід виконання всієї програми буде порушений і в подальшому непередбачуваний. Таких ситуацій слід уникати і щоб їх не допустити рекомендується використовувати якомога менше вкладених функцій і тимчасових змінних, а також не гаяти ОЗУ, втім брак оперативки одвічна проблема .

Регістри введення / виводу (**I / O registers**) - за допомогою регістрів введення / виведення реалізується настройка і управління периферійними пристроями мікроконтролера. При написанні програми, регістрами вводу / виводу можна користуватися як звичайними змінними. У більшості периферійних пристроїв є регістр контролю (Control Register), за допомогою якого можна налаштувати режим роботи периферійного пристрою, а також регістр стану, за допомогою якого можна стежити за ходом роботи периферійного пристрою.

Під регістри відведені молодші 96 (256) адрес. Кожен регістр має свою власну адресу в просторі пам'яті даних. Тому до регістрів можна звертатися двома способами - як до регістрів і як до пам'яті, незважаючи на те, що фізично ці регістри не є осередками ОЗУ. Так, регістрів загального призначення R0 - R31 відповідають адреси ОЗУ $ 000 - $ 01F, регістрів введення / виведення $ 00 - $ 3F відповідають адреси ОЗУ $ 020 - $ 05F (номер регістра плюс $ 20). Таблиця 1. Регістри введення / виводу мікроконтролера ATmega

Название	Адрес	Функция
SREG	$3F ($5F)	Регистр состояния
SPH	$3E ($5E)	Указатель стека, старший байт
SPL	$3D ($5D)	Указатель стека, младший байт
OCR0	$3C ($5C)	Регистр совпадения таймера/счетчика ТО
GICR	$3B ($5B)	Общий регистр управления прерываниями
GIFR	$3A ($5A)	Общий регистр флагов прерываний
TIMSK	$39 ($59)	Регистр маски прерываний от таймеров/счетчиков
TIFR	$38 ($58)	Регистр флагов прерываний от таймеров/счетчиков
SPMCR	$37 ($57)	Регистр управления и состояния операций записи в память программ
TWCR	$36 ($56)	Регистр управления TWI
MCUCR	$35 ($55)	Регистр управления микроконтроллера
MCUCSR	$34 ($54)	Регистр управления и состояния микроконтроллера
TCCR0	$33 ($53)	Регистр управления таймера/счетчика ТО

TCNT0	$32 ($52)	Счетный регистр таймера/счетчика TO
OSCCAL	$31 ($51)	Регистр калибровки тактового генератора
SFIOR	$30 ($50)	Регистр специальных функций
TCCR1A	$2F ($4F)	Регистр А управления таймера/счетчика Т1
TCCR1B	$2E ($4E)	Регистр В управления таймера/счетчика Т1
TCNT1H	$2D ($4D)	Счетный регистр таймера/счетчика Т1, старший байт
TCNT1L	$2C ($4C)	Счетный регистр таймера/счетчика Т1, младший байт
OCR1AH	$2B ($4B)	Регистр А совпадения таймера/счетчика Т1, старший байт
OCR1AL	$2A ($4A)	Регистр А совпадения таймера/счетчика Т1, младший байт
OCR1BH	$29 ($49)	Регистр В совпадения таймера/счетчика Т1, старший байт
OCR1BL	$28 ($48)	Регистр В совпадения таймера/счетчика Т1, младший байт
ICR1H	$27 ($47)	Регистр захвата таймера/счетчика Т1, старший байт
ICR1L	$26 ($46)	Регистр захвата таймера/счетчика Т1, младший байт
TCCR2	$25 ($45)	Регистр управления таймера/счетчика Т2
TCNT2	$24 ($44)	Счетный регистр таймера/счетчика Т2
OCR2	$23 ($43)	Регистр совпадения таймера/счетчика Т2
ASSR	$22 ($42)	Регистр состояния асинхронного режима
WDTCR	$21 ($41)	Регистр управления сторожевого таймера
UBRRH	$20 ($40)	Регистр скорости передачи USART, старший байт
UCSRC		Регистр управления и состояния USART

Окончание табл. 1

Название	Адрес	Функция
EEARH	$1F ($3F)	Регистр адреса EEPROM, старший байт
EEARL	$1E ($3E)	Регистр адреса EEPROM, младший байт
EEDR	$1D ($3D)	Регистр данных EEPROM
EECR	$1C ($3C)	Регистр управления EEPROM
PORTA	$1B ($3B)	Регистр данных порта А
DDRA	$1A ($3A)	Регистр направления данных порта А
PINA	$19 ($39)	Выводы порта А
PORTB	$18 ($38)	Регистр данных порта В
DDRB	$17 ($37)	Регистр направления данных порта В
PINB	$16 ($36)	Выводы порта В
PORTC	$15 ($35)	Регистр данных порта С
DDRC	$14 ($34)	Регистр направления данных порта С
PINC	$13 ($33)	Выводы порта С
PORTD	$12 ($32)	Регистр данных порта D
DDRD	$11 ($31)	Регистр направления данных порта D
PIND	$10 ($30)	Выводы порта D
SPDR	$0F ($2F)	Регистр данных SPI
SPSR	$0E ($2E)	Регистр состояния SPI
SPCR	$0D ($2D)	Регистр управления SPI
UDR	$0C ($2C)	Регистр данных USART
UCSRA	$0B ($2B)	Регистр А управления и состояния USART
UCSRB	$0A ($2A)	Регистр В управления и состояния USART
UBRRL	$09 ($29)	Регистр скорости передачи USART, младший байт
ACSR	$08 ($28)	Регистр управления и состояния аналогового компаратора
ADMUX	$07 ($27)	Регистр управления мультиплексором АЦП
ADCSRA	$06 ($26)	Регистр А управления и состояния АЦП
ADCH	$05 ($25)	Регистр данных АЦП, старший байт
ADCL	$04 ($24)	Регистр данных АЦП, младший байт
TWDR	$03 ($23)	Регистр данных TWI
TWAR	$02 ($22)	Регистр адреса TWI

TWSR	$01 ($21)	Регистр состояния TWI
TWBR	$00 ($20)	Регистр скорости передачи TWI

У ОЗУ даних може бути організований стек, для використання якого необхідно проинициализировать покажчик стека (пару регістрів введення / виводу SPH: SPL). Запис в стек виконується в бік зменшення адрес пам'яті. В стек автоматично заноситься адреса повернення при виклику підпрограми або генерації переривання, а також можна програмно записати і вважати будь-яку інформацію за допомогою команд занесення в стек (PUSH) та вилучення з стека (POP).

Для довготривалого зберігання різної інформації, яка може змінюватися в процесі функціонування готової системи (калібрувальні константи, серійні номери, ключі і т. П.), В мікроконтролерах сімейства може використовуватися вбудована EEPROM-пам'ять. Її обсяг становить для різних моделей від 256 байт до 4 Кбайт. Ця пам'ять розташована в окремому адресному просторі, а доступ до неї здійснюється за допомогою трьох регістрів введення / виводу: регістра адреси, регістра даних і регістра управління.Регістр адреси EEAR (EEPROM Address Register) фізично розміщується в двох PBB - EEARH: EEARL. В цей регістр завантажується адреса комірки, до якої буде проводитися звернення. Регістр адреси доступний як для запису, так і для читання. Причому в регістрі EEARH використовуються тільки молодші біти (кількість задіяних бітів залежить від обсягу EEPROM-пам'яті). Незадіяні біти регістра EEARH доступні тільки для читання і містять нулі.Під час запису в регістр даних EEDR (EEPROM Data Register) заносяться значення, які будуть збережені в EEPROM за адресою, що визначається регістром EEAR, а під час читання в цей регістр поміщаються дані, лічені з EEPROM.Регістр управління EECR (EEPROM Control Register) використовується для управління доступом до EEPROM-пам'яті.

У розділі "**Instruction Set Summary**" кожного даташіта, можна знайти опис всіх асемблерних інструкцій, в тому числі які біти регістра стану (Flags) вони можуть змінити, за скільки тактів виконується кожна інструкція (Clocks).

Instruction Set Summary

Mnemonics	Operands	Description	Operation	Flags	#Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H	1
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2
IJMP		Indirect Jump to (Z)	PC ← Z	None	2
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None	3
ICALL		Indirect Call to (Z)	PC ← Z	None	3
RET		Subroutine Return	PC ← STACK	None	4
RETI		Interrupt Return	PC ← STACK	I	4
PUSH	Rr	Push Register on Stack	STACK ← Rr	None	2
POP	Rd	Pop Register from Stack	Rd ← STACK	None	2

Мал. 4 Приклад асемблерних інструкцій

На рис.4 наведено кілька ассемблерних інструкцій з яких можна помітити що команди розгалуження (Branch instructions) з впливають на регістр стану (крім команди reti), а також ці інструкції виконуються за більше число тактів ніж інші.Також тут можна подивитися як працюють лічильник команд (PC) і покажчик стека (STACK).

3. Особливості компіляції та структура програми на мові Сі.

В мові Сі виповнюється тільки той код що міститься в головній функції (main function). Стосовно до мікроконтролерів це не зовсім так. Найпростіша програма написана на мові Сі:

```
int main ( void )
{
                                                    return 0 ;
}
```

Розглянемо структуру найпростішої програми і для цього створюємо проект в будь-якому середовищі розробки, наприклад в AVR Studio, вставляємо "порожню функцію" і дивимося: Використовується версія AVR Studio v4.19.Щоб писати програми на мові Сі вам знадобиться ще й AVR Toolchain (версія 3.3), куди входить весь інструментарій необхідний для створення програм на мові Сі (компілятор, редактор зв'язків, і т.д.).

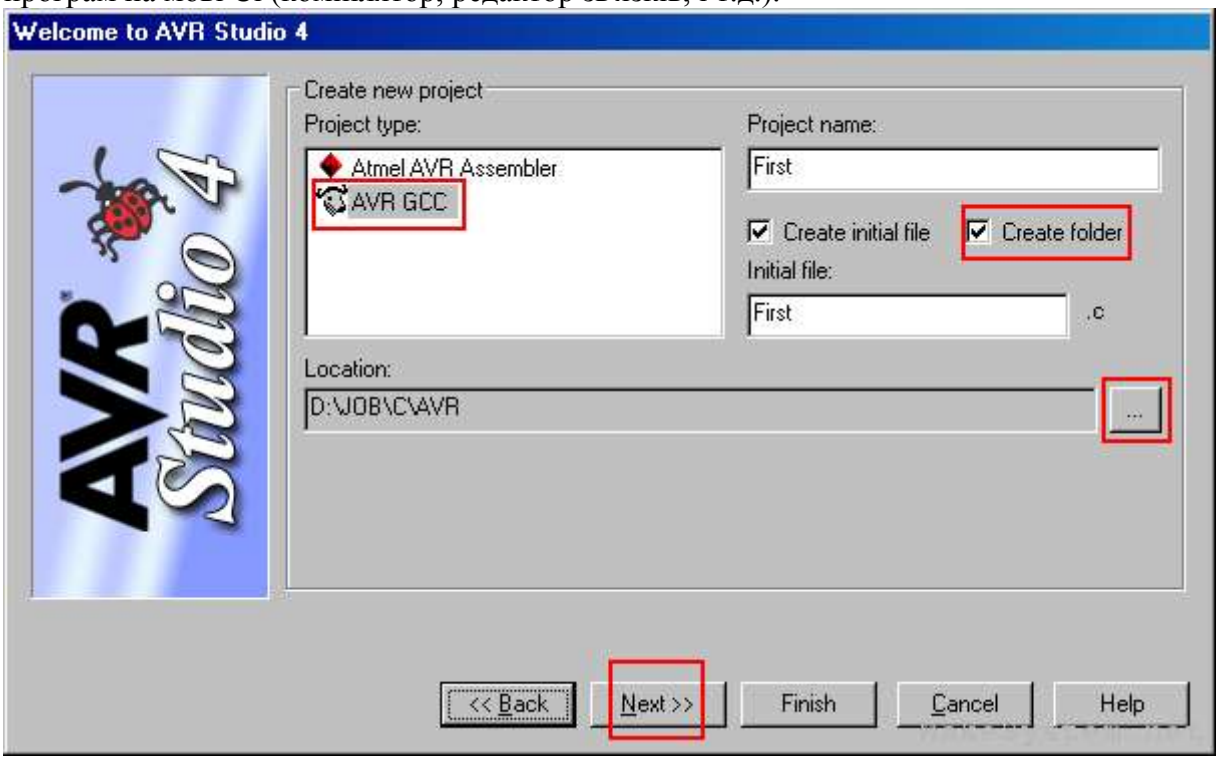


Рис.1 Новий проект в AVR Studio

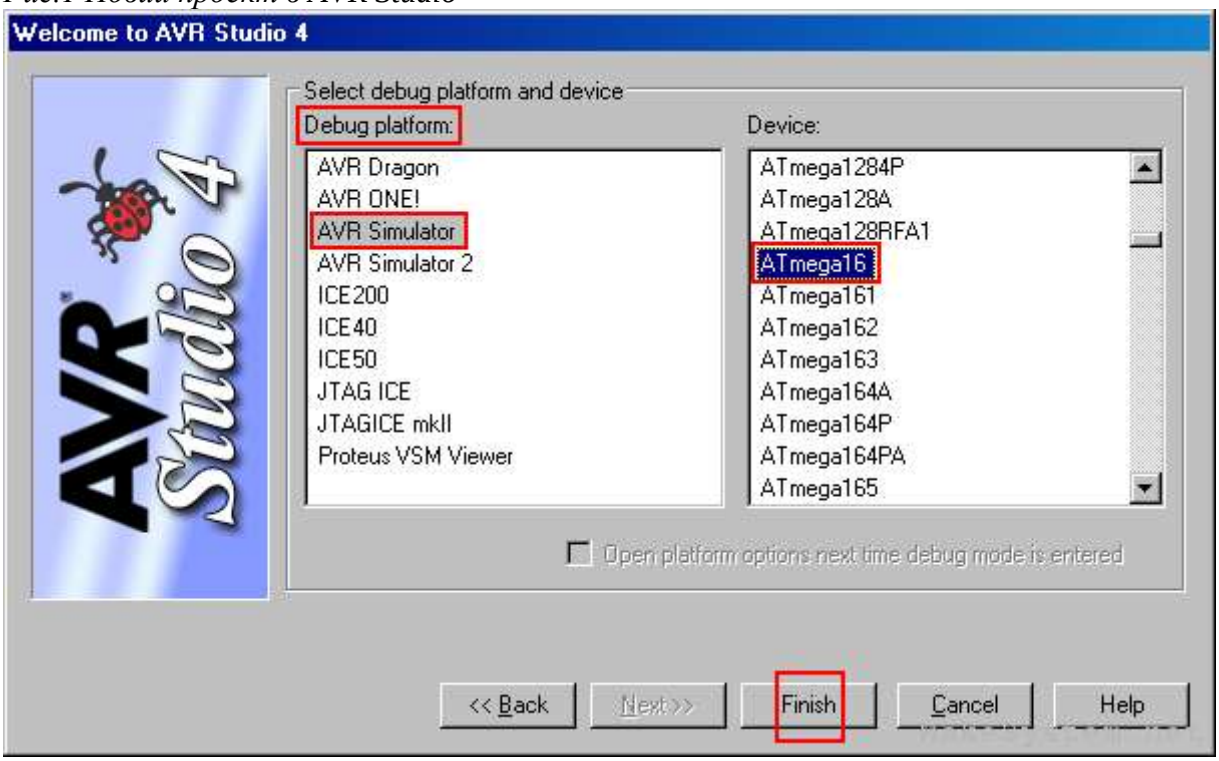


Рис.2 Вибір мікроконтролера

Вибираємо відлагоджувальну платформу (для легких проектів підійде і AVR Simulator) і кристал (ATmega16, з 16.0КБ Flash пам'яті і 1.0КБ ОЗУ, ну а периферію можете подивитися на наступній картинці в вікні I / O View).

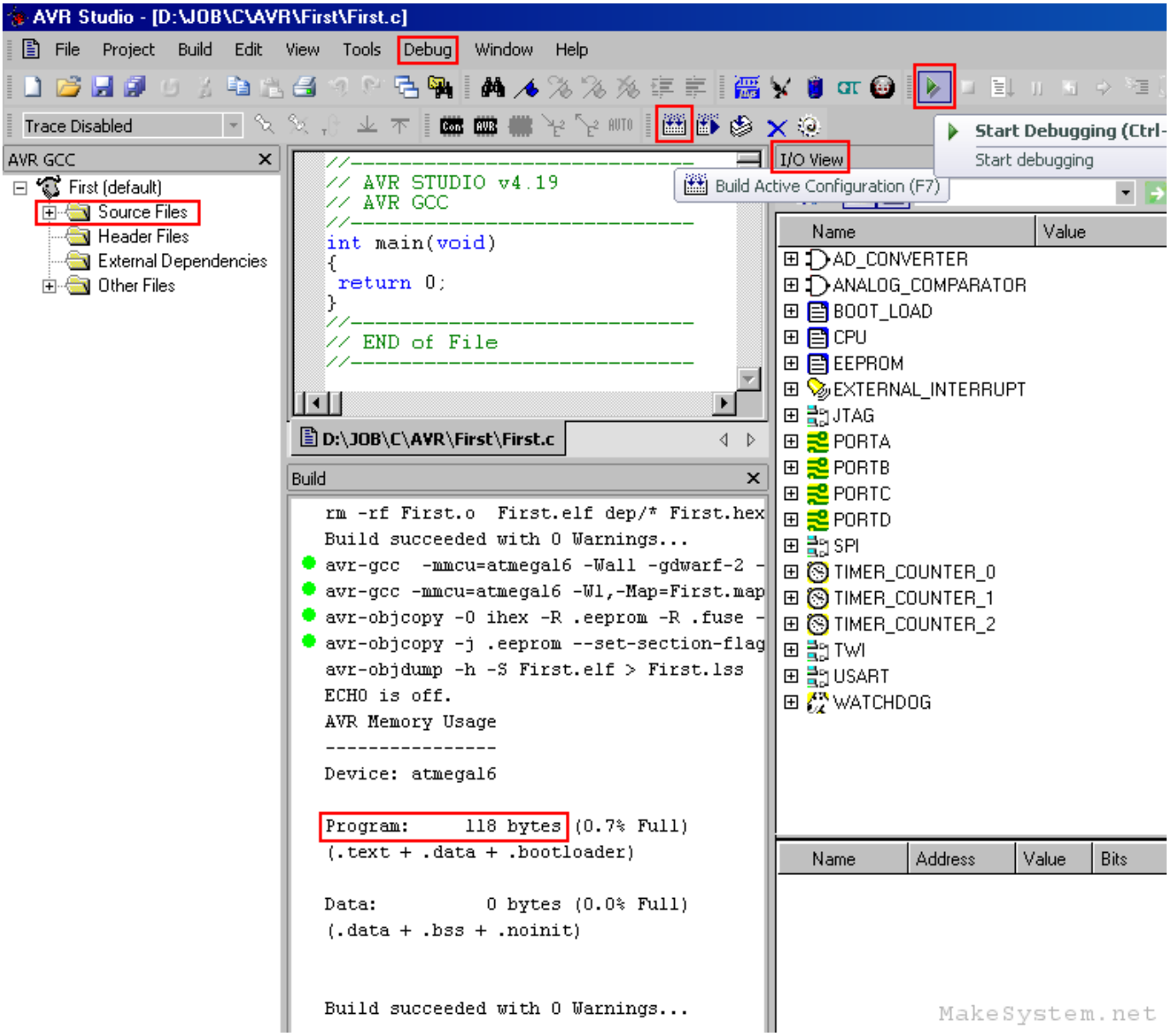


Рис.3 Середовище розробки AVR Studio

Компілюємо наш простий проект і порожня функція буде розміром 118 байт. Для більш детального перегляду вмісту нашого проекту, запускаємо налагодження зайшовши в меню ***Debug-> Start Debugging*** або натиснувши на "зелену стрілку" на панелі швидкого доступу. Далі відкриваємо вікно дизассемблера (меню ***View-> Disassembler***) і спостерігаємо таку картину:

Рис.4 Вікно дизассемблера

У першій колонці вікна дизассемблера, вказані адреси в пам'яті програм де зберігаються асемблерні команди, у другій - кодування кожної команди, в третій - назва ассемблерної команди, в четвертій - операнди кожної інструкції, ну а в п'ятій - короткий опис (що робить кожна команда) .

Як видно з коду, перші 21 команди в програмі займають посилання на різні ділянки коду (інструкція JMP). З рис.4 (перший стовпчик) видно що інструкція JMP займає не 2 байта як більшість асемблерних інструкції а цілих 4 (2 words), що дозволяє посилатися на будь-яку область пам'яті програм. Цифра 21 не випадкова, тому-що у ATmega16 є якраз 21 апаратних переривань (наприклад у ATmega128 - аж 35 векторів переривань). За кожним перериванням жорстко закріплений певний адреса в пам'яті програм. Всі разом, ці "посилання на підпрограми" по обробки переривань (вектори переривань), утворюють так звану "таблицю переривань" (vector table). Пріоритет кожного переривання убуває з просуванням по таблиці переривань. Переривання з найбільшим пріоритетом розташована за адресою 0 × 00000000 (нуль), тобто найперша інструкція яку виконає мікроконтролер це відправиться на обробку переривання, а саме переривання **RESET** (Reset interrupt). Наступне переривання має пріоритет менше ніж у RESET, але більше ніж у наступного переривання і т.д.

Отже, давайте розглянемо більш детально що робить наша марна функція. Перша команда яку виконає мікроконтролер розташована за адресою 0 (нуль) пам'яті програм:

0 × 00000000 **JMP 0x0000002A** // перехід до адресою 0x0000002A в пам'яті програм

Далі, починаючи з адреси 0x0000002A і до адреси 0 × 00000030 йде послідовність команд (в синьому прямокутнику на рис.4) , " **ініціалізація мікроконтролера** " і тільки в кінці цієї самої підпрограми викликається наша "порожня" функція main.

Ініціалізація починається з обнулення регістра стану (Status register), що також має на увазі і відключення всіх переривань.

```
0x0000002A  CLR  R1 // скидання регістра R1 (обнуляем регістр)
0x0000002B  OUT  0x3F, R1 // обнуляем регістр стану процесора
```

Далі йде ініціалізація стека, а якщо точніше покажчика стека (Stack pointer), шляхом завантаження початкової адреси (дно ОЗУ пам'яті) в покажчик стека.

```
0x0000002C  LDI  R28,  0x5F // готуємо молодший байт адреси
0x0000002D  LDI  R29,  0 × 04 // готуємо старший байт адреси
0x0000002E  OUT  0x3E,  R29 // записує старший байт адреси дна ОЗУ пам'яті
0x0000002F  OUT  0x3D, R28 // записує молодший байт адреси дна ОЗУ пам'яті
```

Мікроконтролер по черзі виконує всі ці інструкції, потім викликає функцію main.

```
// Виклик функції main
0 × 00000030  CALL  0 × 00000036 // виклик функції розташованої за адресою 0 × 00000036
```

Після переходу до алресу 0 × 00000036 починається виконання функції main.

```
// Всі що повинна робити наша функція це повернути 16-бітний нуль (тип int)
0 × 00000036  LDI  R24,  0 × 00 // завантажуємо молодший байт нуля
0 × 00000037  LDI  R25,  0 × 00 // завантажуємо старший байт нуля
0 × 00000038  RET // повертаємося і виконуємо наступну інструкцію після виклику функції main
```

Оскільки невідомо кому повинна повернути нуль функція main, то процесор просто поклав нуль в реєстрову пару [R24, R25] і повернувся (RET) для продовження виконання програми. Внаслідок того що функція main завершилася, далі йде стрибок до чергової вставці компілятора:

```
0 × 00000032  JMP  0 × 00000039
// після стрибка
0 × 00000039  CLI  // відключити всі переривання
0x0000003A  RJMP PC-0 × 0000 // стрибаємо на місці
```

Оскільки ми дійшли до кінця нашої програми, то компілятор організував свого роду "заглушку" (вічний цикл), тобто він відключив усі переривання і почав "скакати на місці" (більше не буде виконано жодної іншої команди).

Всі інші вектори переривань (порожні) вказують на одну і ту ж рядок коду, а вона в свою чергу вказує на першу інструкцію пам'яті програм (адреса 0 × 00000000):

```
0 × 00000002  JMP  0 × 00000034
0 × 00000004  JMP  0 × 00000034
0 × 00000006  JMP  0 × 00000034
0 × 00000008  JMP  0 × 00000034
0x0000000A  JMP  0 × 00000034
0x0000000C  JMP  0 × 00000034
0x0000000E  JMP  0 × 00000034
0 × 00000010  JMP  0 × 00000034
0 × 00000012  JMP  0 × 00000034
0 × 00000014  JMP  0 × 00000034
0 × 00000016  JMP  0 × 00000034
0 × 00000018  JMP  0 × 00000034
0x0000001A  JMP  0 × 00000034
0x0000001C  JMP  0 × 00000034
```

```
0x0000001E JMP 0 × 00000034
0 × 00000020 JMP 0 × 00000034
0 × 00000022 JMP 0 × 00000034
0 × 00000024 JMP 0 × 00000034
0 × 00000026 JMP 0 × 00000034
0 × 00000028 JMP 0 × 00000034
// всі вони переходять за адресою 0 × 00000034
// де в свою чергу знаходиться інший перехід
0 × 00000034 JMP 0 × 00000000
```

Таким чином якщо раптом відбудеться переривання, то воно призведе до першої комірки пам'яті програм і тоді мікроконтролер почне виконувати програму заново.

Варто також зауважити, що в тих проектах, де використовуються глобальні змінні, константи, масиви і т.д., а це переважна більшість, є ще одна невелика але дуже важлива підпрограма, яка реалізує **ініціалізацію змінних** .Вона зазвичай йде відразу після ініціалізації стека, а оскільки в нашому проекті не міститься жодних змінних то і форматувати нічого. Всі ці функції ініціалізації (стека, змінних, векторів переривань і т.д.), зазвичай об'єднані в асемблерному файлі під назвою AVRstartup.s або crtavr.s або ще як-небудь.

Цей невеликий приклад наочно показує як редактор зв'язків (компоновщик), компілятор і інші утиліти організують початок (ініціалізацію) і завершення роботи AVR мікроконтролера.Тобто нам з вами залишається тільки написати функцію main і обробники переривань (легко сказати), а іншим займуться утиліти. Така організація Сі-проекту проста, надійна і напрошується сама собою, і фактично є стандартом для всіх розробників Сі-компіляторів більшості сучасних мікроконтролерів (AVR в тому числі).

пишемо програму

Як ми з вами переконалися, перед завершенням, функція main повертає значення "в нікуди", після чого входить в вічний цикл (щоб уникнути непотрібних дій), де більше не виконає жодної команди. Наступна конструкція, практично не відрізняється від розглянутої вище, але як мінімум більш раціональна і інформативна по відношенню до мікроконтролерів.

```
void main ( void )
{
// ----------------------------------
// ініціалізація

наше додаток
// ----------------------------------
}
}
```

Така конструкція нічого не повертає, а також, розділяє програму на дві ділянки: ініціалізація периферії і робочий цикл. Сенс такого поділу в тому, що перша ділянка виповнюється тільки один раз, а другий циклічно - раз по раз. Такий поділ, робить програму більш організованою і зрозумілою.

Оскільки ініціалізація периферії, оголошення і ініціалізація глобальних змінних і т.д., зазвичай має виробляється тільки один раз і до початок роботи з ними (до входу в робочий цикл), то перша ділянка функції main якраз і використовується для цієї мети. Також, щоб можна було скористатися переривань, до входу в цикл слід вирішити всі переривання, за допомогою регістра стану.

Робочий цикл призначений для постійного виконання призначеного для користувача коду: рішення будь-яких вирівняна, обробка даних, передача / прийом даних і так далі, на скільки вистачить вашої фантазії.

Щоб ви могли зробити ініціалізацію периферії, а також використовувати переривання, вам знадобитися підключити до проекту кілька заголовків файлів. У заголовних файлах перебувати опис периферійних регістрів, регістра стану, покажчика стека, бітів кожного регістру, векторів переривань а також багато корисних макроозначень. Ці файли дозволяють розробнику працювати з регістрами периферії як зі звичайними змінними, тим самим забезпечуючи певний рівень абстракції. Навіть в асемблері є заголовки з описом регістрів.Файли підключаються за допомогою препроцесорну директив "**#include** ". Якщо підключити файл " **io.h** ", то AVR Studio визначить (з налаштування проекту) який мікроконтролер ми використовуємо і підключить для даного проекту відповідний заголовки.

У AVR GCC компілятора є одна особливість, він вважає за краще щоб функція main возвращяла значення цілого типу (int). В іншому випадку, компілятор генерує попередження . Нижче наведено приклад, підключення заголовних файлів (для використання периферії і переривань) в AVR Studio і ImageCraft IDE.

Приклад для AVR Studio:

```
#include <avr/io.h>
#include <avr/interrupt.h>

int main ( void )
{
    unsigned global_data ; // всякі змінні
    peripheral_initialization ( ) ; // ініціалізація периферії
    sei ( ) ; // дозволяємо все переривання

    while ( 1 )
    {
        super_code ( ) ; // ваша програма
    }

    return 0 ; // щоб не було warning'ов
}
```

Приклад для ImageCraft IDE:

```
#include <iom16v.h> // ATmega16

void main ( void )
{
    unsigned global_data ; // всякі змінні
    peripheral_initialization ( ) ; // ініціалізація периферії
    sei ( ) ; // дозволяємо все переривання

    while ( 1 )
    {
        super_code ( ) ; // ваша програма
    }
}
```

де ініціали "io **m16** v.h" відповідають микроконтроллеру ATmega16, "io **m128** v.h" - ATmega128, "io **t13** v.h" - ATtiny13 і т.д.

4. Операнди й операції мови Сі.

Операції. По кількості операндов, що беруть участь в операції, операції підрозділяються на унарні, бінарні й тернарні [4, 5].

У мові Сі є наступні унарні операції:

«-» - арифметичне заперечення (заперечення й доповнення);

«~» - побітове логічне заперечення (доповнення);

«!» – логічне заперечення;

«*» - розадресація (непряма адресація);

«&» - обчислення адреси;

«+» - унарний плюс;

«++» - збільшення (інкремент);

«--» - зменшення (декремент);

«sizeof» - розмір.

Унарніопераціі виконуються праворуч ліворуч.

Операції збільшення й зменшення збільшують або зменшують значення операнда на одиницю й можуть бути записані як праворуч так і ліворуч від операнда. Якщо знак операції записаний перед операндом (префіксна форма), то зміна операнда відбувається до його використання у виразі. Якщо знак операції записаний після операнда (постфіксная форма), то операнд спочатку використається у виразі, а потім відбувається його зміна.

Бінарні операції, на відміну від унарних (їхній список наведений у табл. 2.3), виконуються ліворуч праворуч.

Бінарні операції в мові Сі

Таблиця 2.3

Знак операції	Операція	Група операцій
*	множення	мультиплікативні
/	розподіл	
%	залишок від розподілу	
+	додавання	адділтивні
-	віднімання	
<<	зсув вліво	операції зсуву
>>	зсув вправо	
<	менше	операції відносини
>	більше	
<=	менше або дорівнює	
>=	більше або дорівнює	
==	дорівнює	
!=	не дорівнює	
&	поразрядне И	поразрядні
|	поразрядне АБО	
^	поразрядне виключаєче АБО	
&&	логічне И	логічні
||	логічне АБО	
,	послідовне обчислення	послідовного обчислення
=	присвоювання	
*=	множення із присвоюванням	
/=	ділення із присвоюванням	

%=	залишок від ділення із присвоюванням	операції присвоювання
-=	віднімання із присвоюванням	
+=	додавання із присвоюванням	
<<=	зсуву вліво із присвоюванням	
>>=	зсуву вправо із присвоюванням	
&=	поразрядное И с присвоюванням	
|=	поразрядное АБО із присвоюванням	
^=	поразрядное що виключає АБО із присвоюванням	

Лівий операнд операції присвоювання повинен бути виразом, що посилається на область пам'яті (але не об'єктом, оголошеним із ключевим словом const). Такі вираз називаються ліводопустимим, до яких відносяться:

/ідентифікатори даних цілого й плаваючого типів, типів вказівника, структури, об'єднання;

/індексні вираз, крім виражень, що мають тип массива або функції;

/вираз вибору елемента «->» і «.», якщо обраний елемент є ліводопустимим;

/вираз унарної операції розадресації «*», за винятком виражень, що посилаються на масив або функцію;

/вираз приведення типу, якщо результуючий тип не перевищує розміру первісного типу.
При записі виражень варто пам'ятати, що символи «*», «&», «!»,
«+» можуть позначати унарну або бінарну операцію.

2.1. Основні категорії операторів мови Сі

Всі оператори мови Сі можуть бути умовно підрозділені на наступні категорії:

- умовні оператори, до яких ставляться оператор умови if і оператор вибору switch;
операторы циклу (for, while, do while);
операторы переходу (break, continue, return, goto);
- інші оператори (оператор «вираз», порожній оператор).

Оператори в програмі можуть поєднуватися в складені оператори за допомогою фігурних дужок. Будь-який оператор у програмі може бути позначений міткою, що складається з імені й наступного за ним двокрапки.

Всі оператори мови Сі, крім складених операторів, закінчуються крапкою з коми «;» [4, 5].

ADC_INT - по событию "окончание АЦ преобразования"

*/

interrupt [ADC_INT] void adc_isr(void)
{
PORTB=(unsigned char) (~(ADCW>>2));
/* отобразить горящими светодиодами подключенными
от + питания МК через резисторы 560 Ом к ножкам порта_B старшие 8 бит результата аналого-цифрового преобразования

Сделаем паузу 127 мСек - просто как пример пауз */
delay_ms(127);

/*
В реальных программах старайтесь
не делать пауз в прерываниях !

Обработчик прерывания должен быть
как можно короче и быстрее.

Например - в обработчике прерывания вы только устанавливаете флаги (биты или переменная) означающие состояние кнопок, значения переменных или регистров, а обрабатываете это уже в основном цикле программы, через конструкции if - else или switch (описаны выше!)

*/

// начать новое АЦПреобразование
ADCSRA|=0x40;

} // закрывающая скобка обработчика прерывания

Функция обработчик прерывания может быть названа
вами произвольно - как и любая функция кроме main.

Здесь она названа **:** **adc_isr**

При каком прерывании ее вызывать - компилятор узнает из строчки :

interrupt[ADC_INT]

по первому зарезервированному слову - interrupt - он узнаёт,
что речь идет об обработчике прерывания,

а номер вектора прерывания (адрес куда физически, внутри МК перескочит программа при возникновении прерывания) будет подставлен вместо ADC_INT препроцессором компилятора перед компиляцией - этот номер указан в подключенном нами ранее

заголовочном файле ("хидере") описания "железа" МК - **mega16.h** - это число сопоставленное слову **ADC_INT**. Не ленитесь, посмотрите в файле !

Очень информативна и тем ценна для
обучающегося следующая строка программы:

PORTB = (unsigned char) (~(ADCW >> 2));

Давайте проанализируем как она работает.

= оператор присваивания. Он означает присвоить значение вычисления выражения справа от оператора присваивания той переменной что указана слева от него.

Значит нужно вычислить выражение справа и поместить его в переменную PORTB.

Вычислим что справа от оператора присваивания.

ADCW - это переменная слово (двухбайтовая величина - так она объявлена в файле mega16.h) в котором CodeVisionAVR сохраняет 10-битный результат АЦП - а именно в битах9_0 (биты с 9-го по 0-й) т.е. результат выровнен обычно - вправо.

VMLAB имеет только 8 светодиодов - значит нужно отобразить 8 старших бит результата - т.е. биты_9_2 - для этого мы сдвигаем все биты слова ADCW вправо на 2 позиции

ADCW >> 2 /* биты 1 и 0 вылетают вправо из числа в небытие,
бит_9 перемещается в позицию бит_7, бит_8 в позицию бит_6 и
так далее до бит_2 становится бит_0 */

Теперь старшие 8 бит результата АЦП встали в биты7_0
младшего байта (LowByte - LB) слова **ADCW**

>> n означает сдвинуть все биты числа вправо на n позиций

это равносильно делению на 2 в сепени n

<< n

означает сдвинуть все биты числа влево на n позиций

это равносильно умножению на 2 в сепени n

Сдвиг используется очень часто !

Светодиоды подключены так как написано выше - т.е. подключены правильно !

Загораются (показывая "1") при "0" на соответствующем выводе МК - значит нам нужно выводить в PORTB число в котором "1" заменены "0" и наоборот -
это делает как я рассказал выше:

~ операция побитного инвертирования - меняет значения битов.

Значит результатом этого выражения

~(ADCW >> 2)

будут инвертированные 8 старших бит результата АЦП находящиеся
в младшем (правом - LB) байте двух байтового слова ADCW

Выше я уже говорил что :

в Си в переменную можно помещать только тот тип данных который она может хранить !

Так как PORTB это байт, а ADCW - это два байта, то прежде чем выполнить оператор присваивания (это знак =) нужно преобразовать слово (слово - word - значит два байта) ADCW в без знаковый байт.

Преобразование типов данных - делают так :

перед тем что надо преобразовать записывают в скобках ( )
тип данных к которому нужно преобразовать.

На сам код Сі особливих обмежень немає, все ті ж покажчики, масиви, функції, умови і т.д., однак варто пам'ятати:

1. Не використовувати багато бібліотечних функцій. Вони роздмухують розмір програми.

Іноді краще навіть написати вузькоспеціалізований аналог для якоїсь функції.

2. Відмовитися від динамічного виділення пам'яті. Благо програми для мк рідко на таке робить замах.

3. Пам'ятати, що математичного співпроцесора немає і float емулюються. Відповідно теж сильно роздуває код.

4. Пам'ятати, що архітектура 8 біт і не захоплюватися int і long.

5. Пам'ятати, що пам'яті не багато і не розводити змінних.

6. І ніякого C ++. Однак сам GCC дозволяє писати об'єктно-орієнтована код. Але не ті ресурси, щоб так жирувати.

7. Не захаращувати стек великим числом переданих змінних. іноді краще оголосити щось глобально. Так само не перестаратися в самих функціях, тому що пам'ять для локальних змінних з функції виділяється теж в стеці.

В принципі знати асемблер зовсім не обов'язково. Навіть вся периферія нормально конфигурируется з Сі. Але обов'язково треба розуміти все бітові операції в Сі, такі як накладення масок (установки / скидання бітів) і т.д. Без цього периферія не піддасться.

Про конфігурацію периферії можна помітити, що GCC має якісь бібліотечні функції для роботи з нею, ініціалізації. Але нормальних описів не бачив. Віддаю перевагу розбиратися і конфігурація вручну.

Про ресурси: благо мейкфайл в GCC показує скільки вийшло за обсягом коду і потрібно ОЗУ. Тому можна контролювати скільки ресурсів залишилося.

Далі особливість програмування, пов'язана з Гарвардської архітектурою пам'яті. Через те, що шини даних і програм роздільні, Сі-компілятор в коді ініціалізації виробляє переміщення секцій як з пред-ініціалізованих змінними, так і з константами в ОЗУ, щоб до них був безпосередній доступ і нормальна адресація через масиви, покажчики і т.д. Але це може бути дуже незручним, коли є велика таблиця з константами читання з якої здійснюється не дуже інтенсивно. В цьому випадку через спеціальні функції та типи можна витягувати дані не розміщуючи її цілком в ОЗУ.

Лекція 4 Обробка переривань в мікроконтролерах AVR. Таймери.

Питання:

1. **Загальні відомості про переривання**
2. **Прототипи процедури обробки переривання**
3. **Виконання переривання в AVR мікроконтролерів**
4. **Таймери.**

До складу AVR мікроконтролерів входить велика кількість периферійних пристроїв (ADC, Timer / Counters, EXTI, Analog Comparator, EEPROM, USART, SPI, I2C і т.д.), кожне з яких може виконувати певні дії над даними / сигналами тощо. Інформацією . Ці пристрої вбудовані в мікроконтролер для підвищення ефективності програми та зниження витрат при розробці всіляких пристроїв на базі AVR мікроконтролерів.

Процесор спілкується / управляє периферійними пристроями за допомогою регістрів введення / виводу (I / O Registers), які розташовуються в пам'яті даних (Data Memory), що дозволяє використовувати їх як звичайні змінні. У кожного пристрою є свої регістри введення / виводу.

Всі регістри введення / виводу (I / O Registers) можна поділити на три групи: регістри даних, регістри управління і регістри стану.

За допомогою регістрів управління (Control Registers) реалізується настройка пристрою для роботи в тому чи іншому режимі, з певною частотою, точністю і т.д., а за допомогою регістрів даних (Data Registers) зчитується результат роботи даного пристрою (аналого-цифрове перетворення, прийняті дані, значення таймера / лічильника і т.д.). Здавалося б, нічого складного тут немає, включив пристрій, вказав бажаний режим роботи а потім тільки залишається читати дані і використовувати їх в обчисленнях. Все питання полягає в тому коли читати ці самі дані (завершило пристрій роботу або все ще обробляє дані), адже всі периферійні пристрої працюють паралельно з ядром мікроконтролера, та ще й на різних частотах. Постає питання реалізації спілкування і синхронізації між процесором і периферійним пристроєм.

Для реалізації спілкування і синхронізацію пристрою із процесором використовуються "регістри стану" (Status Registers), в яких зберігається поточний стан роботи того чи іншого пристрою. Кожному стану, в якому може перебувати пристрій, відповідає "біт в регістрі стану" (прапор), поточне значення якого, "говорить" про поточний стан даного пристрою або його окремо взятої функції (робота завершена / не завершена, помилка при обробці даних, регістр порожній і т.д.).

Механізм спілкування , між процесором і периферійним пристроєм, реалізується шляхом опитування прапорів (flag polling), що відповідають за ту чи іншу функцію даного пристрою. Залежно від значення того чи іншого прапора (стан пристрою), можна змінювати хід виконання програми (розгалуження). Наприклад :

Перевірка якщо певний прапор встановлений (відбулося якесь подія):

```
if ( RegX & ( 1 << Flag ) ) // якщо прапор в регістрі RegX встановлений
{
// роби щось
}
```

Очікування завершення будь-якого дії (подія):

```
while ( ! ( RegX & ( 1 << Flag ) ) ) ; // поки прапор не встановлений - чекаємо
```

Опитування прапорів - заняття досить ресурсномістке, як в плані розміру програми, так і в плані швидкодії програми.Оскільки загальне число прапорів в AVR мікроконтролери досить велике (перевага), то реалізація спілкування, між процесором і пристроєм, шляхом опитування прапорів призводить до зниження ККД (швидкодія коду / розмір коду) написаної

вами програми, до того ж програма стає дуже заплутаною, що сприяє появі помилок, які важко виявити навіть при детальній налагодженні коду.

Для того щоб підвищити ККД програм для AVR мікроконтролерів, а також полегшити процес створення і налагодження даних програм, розробники забезпечили всі периферійні пристрої "джерелами переривань" (**Interrupt sources**), у деяких пристроїв може бути кілька джерел переривання.

За допомогою джерел переривань реалізується **механізм синхронізації** , між процесором і периферійним пристроєм, тобто процесор почне прийом даних, опитування прапорів і ін. Дії над периферійним пристроєм тільки тоді, коли пристрій буде до цього готове (повідомить про завершення обробки даних, помилку при обробці даних , регістр порожній, і т.д.), шляхом генерації "запиту на обробку переривання" (**Interrupt request**), в залежності від значення деякого прапора (стан пристрою / функції / події).

У літературі, дуже часто, весь ланцюжок подій, починаючи від "запиту на обробку переривання" (IRQ) і до "процедури обробки переривання" (ISR), скорочено називають - переривання (**Interrupt**).

Переривання (Interrupt) - *сигнал, що повідомляє процесору про настання якої-небудь події. При цьому виконання поточної послідовності команд припиняється і керування передається процедурі обробки переривання, відповідна даної події, після чого виконання коду триває рівно з того місця де він був перерваний (повернення управління)* .

Процедура обробки переривання (Interrupt Service Routine) - це ні що інше як функція / підпрограма, яку слід виконати при виникненні певної події. Будемо використовувати саме слово "процедура", для того щоб підкреслити її відмінність від всіх інших функцій.

Головна відмінність процедури від простих функцій полягає в тому що замість звичайного "повернення з функції" (асемблерна команда RET), слід використовувати "повернення з переривання" (асемблерна команда RETI) - " **RETurn from Interrupt** ".

Властивості AVR переривань:

- У кожного периферійного пристрою, що входить до складу AVR мікроконтролерів, є як мінімум одне джерело переривання (Interrupt source). До всіх цих переривань слід зарахувати і переривання скидання - Reset Interrupt, призначення якого відрізняється від всіх інших.
- За кожним перериванням, суворо закріплений вектор (посилання) вказує на процедуру обробки переривання (Interrupt service routine). Всі вектори переривань, розташовуються на самому початку пам'яті програм і разом формують "таблицю векторів переривань" (**Interrupt vectors table**).
- Кожному переривання відповідає певний "біт активації переривання" (**Interrupt Enable bit**). Таким чином, щоб використовувати певний переривання, слід записати в його "біт активації переривання" - лог. одиницю. Далі, незалежно від того активували Ви чи ні певні переривання, мікроконтролер не почне обробку цих переривань, поки в "біт загального дозволу переривань" (**Global Interrupt Enable bit** в регістрі стану SREG) нічого очікувати записана лог. одиниця. Також, щоб заборонити всі переривання (на невизначений час), в біт загального дозволу переривань слід записати - лог. нуль.

Переривання Reset, на відміну від всіх інших, не можна заборонити. Такі переривання ще називають Non-maskable interrupts.

SREG – AVR Status Register

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – I: Global Interrupt Enable**

- У кожного переривання є строго певний пріоритет.Пріоритет переривання залежить від його розташування в "таблиці векторів переривань". Чим менше номер

вектора в таблиці, тим вище пріоритет переривання.Тобто, найвищий пріоритет має переривання скидання (Reset interrupt), яке розташовується першої в таблиці, а відповідно і в пам'яті програм. Зовнішнє переривання INT0, що йде слідом за перериванням Reset в "таблиці векторів переривань", має пріоритет менше ніж у Reset, але вище ніж у всіх інших переривань і т.д.

Таблиця векторів переривань, крім вектора Reset, може бути переміщена в початок Boot розділу Flash пам'яті, встановивши біт **IVSEL** в регістрі GICR. Вектор скидання також може бути переміщений в початок Boot розділу Flash пам'яті, шляхом програмування фьюз біта - **BOOTRST** .

Reset and Interrupt Vectors

Vector No.	Program Address	Source	Interrupt Definition
1	$000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	TIMER2 COMP	Timer/Counter2 Compare Match
5	$008	TIMER2 OVF	Timer/Counter2 Overflow
6	$00A	TIMER1 CAPT	Timer/Counter1 Capture Event
7	$00C	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	$00E	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	$010	TIMER1 OVF	Timer/Counter1 Overflow
10	$012	TIMER0 OVF	Timer/Counter0 Overflow
11	$014	SPI, STC	Serial Transfer Complete
12	$016	USART, RXC	USART, Rx Complete
13	$018	USART, UDRE	USART Data Register Empty
14	$01A	USART, TXC	USART, Tx Complete
15	$01C	ADC	ADC Conversion Complete
16	$01E	EE_RDY	EEPROM Ready
17	$020	ANA_COMP	Analog Comparator
18	$022	TWI	Two-wire Serial Interface
19	$024	INT2	External Interrupt Request 2
20	$026	TIMER0 COMP	Timer/Counter0 Compare Match
21	$028	SPM_RDY	Store Program Memory Ready

Рис.1 Таблиця векторів переривань ATmega16

2. Прототипи процедури обробки переривання

Щоб оголосити деяку функцію в якості процедури обробки того чи іншого переривання, необхідно дотримуватися певних правил прототипирования, щоб компілятор / компоновщик змогли правильно визначити і зв'язати потрібне вам переривання з процедурою її обробки.

По-перше процедура обробки переривання не може нічого приймати в якості аргументу (void), а також не може нічого повертати (void). Це пов'язано з тим що всі переривання в AVR асинхронні, тому не відомо в якому місці буде перервано виконання програми, у кого приймати і кому повертати значення, а також для мінімізації часу входу і виходу з переривання.

```
void isr ( void )
```

По-друге, перед прототипом функції слід вказати що вона є процедурою обробки переривання. Як вам відомо, в мові Сі виповнюється тільки той код що використовується в функції**main** . Оскільки процедура обробки переривання в функції main ніде не використовується, то для того щоб компілятор не «викинув" її за непотрібністю, перед прототипом процедури слід вказати що ця функція є процедурою обробки переривання.

Прототип процедури обробки переривання в середовищі AVR Studio

```
#include <avr / interrupt.h>

ISR ( XXX_vect )
{
// Тіло обробника переривання
}
```

У AVR Studio (AVR GCC), кожна процедура обробки переривання починається з макроозначення **ISR** , після чого, в круглих дужках слід конструкція:

```
XXX_vect
```

де "XXX" це ім'я вектора переривання. Всі імена векторів, для певного AVR мікроконтролера, можна знайти в "таблиці векторів переривань" даташіта даного мікроконтролера або в його заголовки. Наприклад, "таблиця векторів переривань" для мікроконтролера ATmega16 приведена на рис.1, де в колонці**Source ,** наведені всі імена векторів переривань. Також імена можна подивитися в заголовки даного мікроконтролера (C: \ Program Files \ Atmel \ AVR Tools \ AVR Toolchain \ avr \ include \ avr \ iom16.h), див. Рис.2. Все що нам треба зробити, це знайти в таблиці ім'я потрібного нам вектора і до нього додати суфікс " **_vect** ".

Далі, в фігурних дужках, пишемо "тіло" процедури обробки даного переривання.

```
/* Interrupt vectors */
/* Vector 0 is the reset vector. */
/* External Interrupt Request 0 */
#define INT0_vect            _VECTOR(1)
#define SIG_INTERRUPT0           _VECTOR(1)

/* External Interrupt Request 1 */
#define INT1_vect            _VECTOR(2)
#define SIG_INTERRUPT1           _VECTOR(2)

/* Timer/Counter2 Compare Match */
#define TIMER2_COMP_vect         _VECTOR(3)
#define SIG_OUTPUT_COMPARE2      _VECTOR(3)

/* Timer/Counter2 Overflow */
#define TIMER2_OVF_vect          _VECTOR(4)
#define SIG_OVERFLOW2            _VECTOR(4)

/* Timer/Counter1 Capture Event */
#define TIMER1_CAPT_vect         _VECTOR(5)
#define SIG_INPUT_CAPTURE1       _VECTOR(5)

/* Timer/Counter1 Compare Match A */
#define TIMER1_COMPA_vect        _VECTOR(6)
#define SIG_OUTPUT_COMPARE1A         _VECTOR(6)

/* Timer/Counter1 Compare Match B */
#define TIMER1_COMPB_vect        _VECTOR(7)
#define SIG_OUTPUT_COMPARE1B         _VECTOR(7)

/* Timer/Counter1 Overflow */
#define TIMER1_OVF_vect          _VECTOR(8)
#define SIG_OVERFLOW1            _VECTOR(8)

/* Timer/Counter0 Overflow */
#define TIMER0_OVF_vect          _VECTOR(9)
#define SIG_OVERFLOW0            _VECTOR(9)
```


Рис.2 Заголовний файл ATmega16 для AVR Studio

Для прикладу, напишемо процедуру обробки переривання по прийому байта через USART (USART, Rx Complete):

```
ISR ( USART_RXC_vect )
{
// Тіло обробника переривання
}
```

До речі : перед тим як використовувати будь-яке переривання в AVR Studio, слід включити в проект заголовки **io.h** і **interrupt.h** :

```
#include <avr / io.h>
#include <avr / interrupt.h>
```

Більш докладно про обробниках переривань в AVR Studio (AVR GCC) можна почитати в розділі **Introduction to avr-libc's interrupt handling** .

Прототип процедури обробки переривання в середовищі ImageCraft

```
#pragma interrupt_handler <handler_name>: iv_XXX
void < handler_name > ( void )
{
// Тіло обробника переривання
}
```

У середовищі ImageCraft, прототип процедури обробки переривання виглядає наступним чином:

```
void < handler_name > ( void )
```

де <handler_name>, це будь-яке ім'я яке ви захочете дати даному оброблювачу переривання. Одна з вимог до оголошення процедур обробки переривань свідчить, що перед прототипом функції слід вказати що вона є обробником переривання. Це робиться за допомогою pragma-директиви**interrupt_handler** :

```
#pragma interrupt_handler <handler_name>: iv_XXX
```

де <handler_name> це ім'я тієї функції що буде використовуватися в якості обробника переривання, а конструкція "iv_XXX", це ім'я вектора переривання (XXX) з префіксом " **iv_** ". Як і у випадку з AVR Studio, все імена векторів, для певного AVR мікроконтролера, можна знайти в "таблиці векторів переривань" даташіта даного мікроконтролера або в його заголовки (див. Рис.3).

```
/* Interrupt Vector Numbers */

#define iv_RESET        1
#define iv_INT0         2
#define iv_INT1         3
#define iv_TIMER2_COMP  4
#define iv_TIMER2_OVF   5
#define iv_TIMER1_CAPT  6
#define iv_TIMER1_COMPA 7
#define iv_TIMER1_COMPB 8
#define iv_TIMER1_OVF   9
#define iv_TIMER0_OVF   10
#define iv_SPI_STC      11
#define iv_USART_RX     12
#define iv_USART_RXC    12
#define iv_USART_DRE    13
#define iv_USART_UDRE   13
#define iv_USART_TX     14
#define iv_USART_TXC    14
#define iv_ADC          15
#define iv_EE_RDY       16
#define iv_EE_READY     16
#define iv_ANA_COMP     17
#define iv_ANALOG_COMP  17
#define iv_TWI          18
#define iv_TWSI         18
#define iv_INT2         19
#define iv_TIMER0_COMP  20
#define iv_SPM_RDY      21
#define iv_SPM_READY    21
```


Рис.3 Заголовний файл ATmega16 для ImageCraft IDE

Наприклад процедура обробки переривання по прийому байта через USART (USART, Rx Complete) в середовищі ImageCraft, буде виглядає так:

```
#pragma interrupt_handler usart_rxc_isr: iv_USART_RXC
void usart_rxc_isr ( void )
{
// Тіло обробника переривання
}
```

Більш докладно про процедурах обробки переривання в ImageCraft IDE можна знайти в меню ***Help-> Programming the AVR-> Interrupt Handlers*** середовища розробки.

Іноді, якщо кілька обробників переривання повинні робити одне і те ж, то для економії пам'яті програм, можна направити кілька векторів переривання на одну і ту ж процедуру обробки переривання.

У середовищі AVR Studio це виглядає так:

```
ISR ( INT0_vect )
{
// Do something
}
ISR ( INT1_vect , ISR_ALIASOF ( INT0_vect ) ) ;
```

Спочатку йде процедура обробки переривання для певного вектора, в даному випадку INT0. Всі інші процедури можуть посилатися на будь-який обробник переривання за допомогою конструкції:

```
ISR ( YYY_vect , ISR_ALIASOF ( XXX_vect ) ) ;
```

де YYY це ім'я вектора переривання який посилається на раніше оголошений обробник переривання для вектора XXX.

У середовищі ImageCraft це виглядає так:

```
#pragma interrupt_handler <handler_name>: iv_XXX <handler_name>: iv_YYY
void < handler_name > ( void )
{
// Тіло обробника переривання
}
```

або так

```
#pragma interrupt_handler <handler_name>: iv_XXX
#pragma interrupt_handler <handler_name>: iv_YYY
void < handler_name > ( void )
{
// Тіло обробника переривання
}
```

де вектори XXX і YYY посилаються на один і той же обробник переривання <handler_name>.

3. Виконання переривання в AVR мікроконтролерів

1. Припустимо стався "запит на обробку переривання" (IRQ).

До речі: якщо одночасно відбудуться кілька запитів на обробку переривання, то першим буде оброблено переривання з найвищим пріоритетом, всі інші запити будуть оброблені по завершенню високопріоритетного переривання.

2. **Перевірка** .

Якщо біт активації даного переривання встановлений (Interrupt enable bit), а також I-біт (біт загального дозволу переривань) регістра стану процесора (SREG) встановлено, то процесор починає підготовку процедури обробки переривання, при цьому біт загального дозволу переривань (I-біт регістра SREG) скидається, забороняючи таким чином всі інші переривання.Це відбувається для того щоб ніяка інша подія не змогло перервати обробку поточного переривання.

До речі: якщо в процедурі обробки переривання встановити I-біт в стан лог. одиниці, то будь-який активований переривання може в свою чергу перервати обробку поточного переривання. Такі переривання називаються вкладені (Nested interrupts).

3. **Підготовка .**

Процесор завершує виконання поточної ассемблерной команди, після чого поміщає адресу наступної команди в стек (PC-> STACK). Далі процесор перевіряє яке джерело переривання подав "запит на обробку переривання" (IRQ), після чого скориставшись вектором даного джерела (посилання) з таблиці векторів (який залізно закріплений за кожним джерелом переривання), переходить в процедуру обробки переривання (інструкція JMP). На все, про все процесор витрачає мінімум 4 такту! (В залежності від моменту появи запиту і тривалість виконання поточної інструкції). Це хороший час реакції на IRQ, в порівнянні з мікроконтролерами інших виробників.

До речі: якщо IRQ відбудеться, коли мікроконтролер знаходиться в сплячому режимі (sleep mode), час реакції на IRQ збільшується ще на чотири такту, плюс час закладене в фьюз бітах SUT1 і SUT0 (Start-Up Time).

4. **Виконання тіла ISR .**

До речі: більшість Сі-компіляторів, перед початком і завершенням виконання тіла процедури обробки переривання, вставляють додаткові "підпрограми збереження / выдновленн регістрів" загального призначення (РЗП) і регістра стану (SREG). Так що програмісту немає необхідності самостійно зберігати відновлюємо РЗП під час переривань. Просунуті Сі-компілятори можуть вирахувати скільки РЗП знадобляться даному оброблювачу переривання і вставити підпрограми збереження / восстановленія тільки потрібної кількості РЗП.

5. **Повернення управління**

Після того як виконання процедури обробки переривання завершено, процесор витягує адреса повернення з переривання, який він зберіг в стеці (STACK-> PC), додає до покажчика стека значення 2, що відповідає зменшенню стека на два байта, які раніше займав адреса повернення з переривання . Далі процесор встановлює I-біт (біт загального дозволу переривань) регістра стану процесора (SREG). На все, про все процесор витрачає рівно 4 такту. Після цього виконання програми триває рівно з того місця де вона була перервана.

Поради щодо створення ISR

1. Намагайтеся якнайшвидше вийти з процедури обробки переривання, від цього залежить чуйність вашого застосування. Тобто якщо ви відносно довго будите знаходиться в ISR, а в цей момент будуть з'являтися нові запити на обробку переривання, то почнеться накопичення, а потім і втрата нових подій. Як ви самі розумієте таку поведінку бажано уникнути.

2. Обчислення в переривання повинні бути якомога простіше, від цього залежить кількість зберігаються / відновляє РЗП.

Переривання Reset

У переривання Reset є 5 джерел переривання:

- Скидання при включенні живлення (Power-on Reset).Мікроконтролер знаходиться в стані скидання, поки напруга живлення знаходиться нижче порогового значення V_{POT} .
- Зовнішній скидання (External Reset). Мікроконтролер знаходиться в стані скидання, поки на висновок RESET поданий низький рівень.
- Скидання сторожового таймера (Watchdog Reset).Мікроконтролер скидається через деякий період часу, заданий в сторожовому таймері і коли сторожовий таймер включений.
- Скидання при виході за межа (Brown-Out Reset).Мікроконтролер скидається, коли напруга живлення V_{CC}опускається нижче порогового значення V_{BOT} і коли Brown-Out детектор включений.

- JTAG скидання. Мікроконтролер знаходиться в стані скидання, поки в однобітний регістрі Reset Register знаходиться значення лог. одиниця.

Приклад використання переривання

В даному прикладі мікроконтролер, зчитує значення аналогового сигналу на вході АЦП і посилає оброблені дані через USART, зовнішнього пристрою.

```
// AVR Studio v4.19

#include <avr / io.h>
#include <avr / interrupt.h>

/ * макровизначеннями, для роботи з бітами * /
#define BIT (n) (1 << (n))
# define ENABLE (x, n) ((x) | = BIT (n))
#define CHECKBIT (x, n) ((x) & BIT (n))

void port_init ( void )
{
 PORTA = 0x00 ;
 DDRA = 0x00 ; // порт А робимо вхідним
 PORTB = 0x00 ;
 DDRB = 0x00 ;
 PORTC = 0x00 ; // m103 output only
 DDRC = 0x00 ;
 PORTD = 0x00 ;
 DDRD = 0x00 ;
}

// USART initialize
// desired baud rate: 9600
// actual : baud rate: 9615 (0,2%)
// char size: 8 bit
// parity: Disabled
void usart_init ( unsigned baudrate )
{ // ініціалізація USART модуля
 UCSRB = 0x00 ; // disable while setting baud rate
 UCSRA = 0x00 ;
 UCSRC = BIT ( URSEL ) | 0x06 ;
 UBRRL = baudrate ;
 UCSRB = 0x08 ;
}

// ADC initialize
// Conversion time: 104uS
void adc_init ( void )
{ // ініціалізація АЦП модуля
 ADCSRA = 0x00 ; // disable adc
 ADMUX = 0x00 | ( 1 << ADLAR ) ; / * будемо використовувати перший канал АЦП,
як референсу потенціал поданий на висновок AREF,
8-біт розрядність АЦП і рівняння на ліво :) * /
 ACSR = 0x80 ; // вимикаємо аналоговий компаратор, для економії
 ADCSRA = 0xCD ; / * включаємо АЦП і запускаємо одиночне перетворення,
включаємо переривання по закінченню перетворення,
встановлюємо частоту перетворення * /
}
```

```
ISR ( ADC_vect )
{
 while ( ! CHECKBIT ( UCSRA , UDRE ) ) ; // чекаємо поки звільниться буфер
 UDR = ADCH ; // відсилаємо дані
 ENABLE ( ADCSRA , ADSC ) ; // запускаємо нове перетворення
}

// call this routine to initialize all peripherals
void init_devices ( void )
{
 cli ( ) ; // на час ініціалізації периферії, забороняємо все переривання
 port_init ( ) ;
 usart_init ( 25 ) ;
 adc_init ( ) ;
 MCUCR = 0x00 ;
 GICR = 0x00 ;
 TIMSK = 0x00 ; // timer interrupt sources
 sei ( ) ; // дозволяємо назад все переривання
}

int main ( void )
{
 init_devices ( ) ;
 while ( 1 ) // створюємо нескінченний цикл
 {
 }
}
```

Як видно з цього прикладу, в функції main немає ніякого коду крім ініціалізації. Таке програмування ще називають **interrupt driven programming** , тобто всі дії відбуваються виключно в процедурах обробки переривань, в нашому випадку в обробнику переривання АЦП перетворювача.

```
ISR ( ADC_vect )
{
 while ( ! CHECKBIT ( UCSRA , UDRE ) ) ; // чекаємо поки звільниться буфер
 UDR = ADCH ; // відсилаємо дані
 ENABLE ( ADCSRA , ADSC ) ; // запускаємо нове перетворення
}
```

Переривання ADC відбувається по завершенню аналого - цифрового перетворення.

```
while ( ! CHECKBIT ( UCSRA , UDRE ) ) ; // чекає поки звільниться буфер
```

Тут немає ніякого очікування, оскільки в даному випадку, передача даних через USART трохи - трохи швидше ніж А / Ц перетворення, а ця строчка всього лише додаткова перевірка (перестраховка). По-друге, якби ми використовували переривання UDRE, то тоді процесор переходив би в обробник переривання навіть коли А / С дані ще не готові (нічого передавати). По-третє це єдине переривання, тому втрат у нас бути не може, оскільки нове А / Ц перетворення почнеться тільки після завершення обробника старого.

```
ENABLE ( ADCSRA , ADSC ) ; // запускаємо нове перетворення
```

У функції init_devices (), ми використовували два макроозначення cli () і sei (), які призначені для заборони всіх переривань - cli () (clear I -bit in SREG) і дозволу всіх переривань - sei () (**se** t **I** - bit in SREG).

Висновок

Необхідно якомога більше завдань "вішати" на периферійні пристрої, а також використовувати переривання, це підвищить ККД вашої програми. Надмірно не затримуйтеся в переривання, це може привести до зниження чуйності додатки.

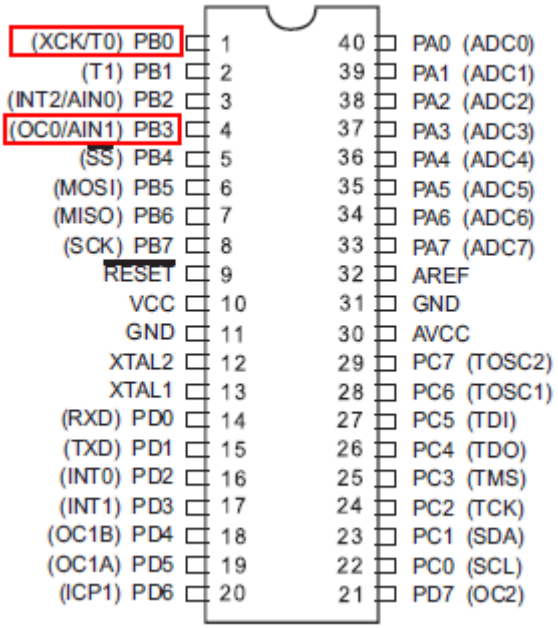
Таймери.

Таймер-лічильник є одним з найбільш ходових ресурсів AVR мікроконтролера. Його основне призначення - відраховувати задані часові інтервали. Крім того, таймери-лічильники можуть виконувати ряд додаткових функцій, як то - формування ШІМ сигналів, підрахунок тривалості і кількості вхідних імпульсів. Для цього існують спеціальні режими роботи таймера-лічильника.

Залежно від моделі мікроконтролера кількість таймерів і набір їх функцій може відрізнятися. Наприклад, у мікроконтролера Atmega16 три таймера-лічильника - два 8-ми розрядних таймера-лічильника T0 і T2, і один 16-ти розрядний - T1. У цій статті, на прикладі ATmega16, ми розберемо як використовувати таймер-лічильник T0.

Використовувані висновки

Таймер-лічильник T0 використовує два висновки мікроконтролера ATmega16. Висновок T0 (PB0) - це вхід зовнішнього тактового сигналу. Він може застосовуватися, наприклад, для підрахунку імпульсів. Висновок OC0 (PB3) - це вихід схеми порівняння таймера-лічильника. На цьому висновку за допомогою таймера може формувати меандр або ШІМ сигнал. Також він може просто міняти свій стан при спрацьовуванні схеми порівняння.



Висновки T0 і OC0 задіюються тільки при відповідних настройках таймера, в звичайному стані це висновки загального призначення.

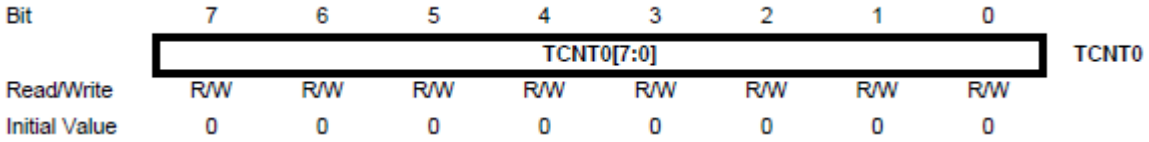
Регістри таймера-лічильника T0

Регістри - це те, без чого неможливо програмувати мікроконтролери. Так ось, таймер T0 має в своєму складі три регістра:

- Рахунковий регістр TCNT0,
- регістр порівняння OCR0,
- конфігураційний регістр TCCR0.

Крім того, є ще три регістра, що відносяться до всіх трьох таймерам ATmega16:
- конфігураційний регістр TIMSK,
- статусний регістр TIFR.
- Регістр спеціальних функцій SFIOR

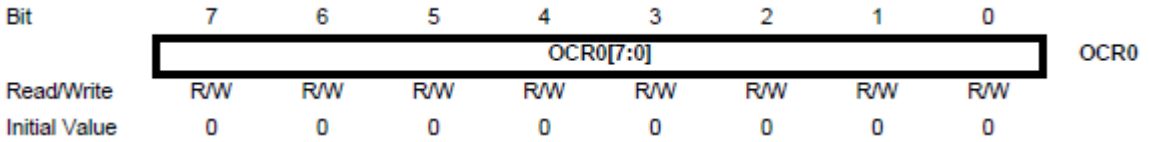
Почнемо з **TCNT0**



Це 8-ми розрядний рахунковий регістр. Коли таймер працює, по кожному імпульсу тактового сигналу значення TCNT0 змінюється на одиницю. В залежності від режиму роботи таймера, рахунковий регістр може або збільшуватися, або зменшуватися.

Регістр TCNT0 можна як читати, так і записувати. Останнє використовується коли потрібно задати його початкове значення. Коли таймер працює, змінювати його вміст TCNT0 не рекомендується, так як це блокує схему порівняння на один такт.

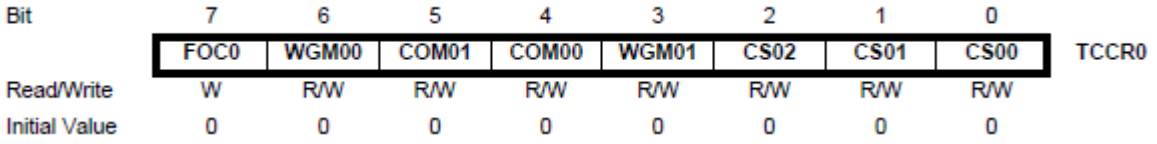
OCR0



Це 8-ми розрядний регістр порівняння. Його значення постійно порівнюється з рахунковим регістром TCNT0, і в разі збігу таймер може виконувати якісь дії - викликати переривання, змінювати стан виведення OC0 і т.д. в залежності від режиму роботи.

Значення OCR0 можна як читати, так і записувати.

TCCR0 (Timer / Counter Control Register)



Це конфігураційний регістр таймера-лічильника T0, він визначає джерело тактирования таймера, коефіцієнт предделителя, режим роботи таймера-лічильника T0 і поведінку виведення OC0. По суті, найважливіший регістр.

Біти **CS02, CS01, CS00 (Clock Select)** - визначають джерело тактової частоти для таймера T0 і задають коефіцієнт предделителя. Всі можливі стани описані в таблиці нижче.

CS02	CS01	CS00	Описание
0	0	0	Источника тактирования нет. Таймер остановлен.
0	0	1	Тактовая частота МК
0	1	0	Тактовая частота МК/8
0	1	1	Тактовая частота МК/64
1	0	0	Тактовая частота МК/256
1	0	1	Тактовая частота МК/1024
1	1	0	Внешний источник на выводе T0. Срабатывание по заднему фронту
1	1	1	Внешний источник на выводе T0. Срабатывание по переднему фронту

Як бачите, таймер-лічильник може бути зупинений, може тактіроваться від внутрішньої частоти і також може тактіроваться від сигналу на виводі T0.

Біти **WGM10, WGM00 (Wave Generator Mode)** - визначають режим роботи таймера-лічильника T0. Всього їх може бути чотири - нормальний режим (normal), скидання таймера при збігу (CTC), і два режими широтно-імпульсної модуляції (FastPWM і Phase Correct PWM). Всі можливі значення описані в таблиці нижче. Більш детально будемо розбирати режими в коді. Зараз всі нюанси все одно не запам'ятаються.

Біти **COM01, COM00 (Compare Match Output Mode)** - визначають поведінку виведення OC0. Якщо хоч один з цих бітів встановлений в 1, то висновок OC0 перестає функціонувати як звичайний висновок загального призначення і підключається до схеми порівняння таймера лічильника T0. Однак при цьому він повинен бути ще налаштований як вихід. Поведінка виведення OC0 залежить від режиму роботи таймера-лічильника T0. У режимах normal і CTC висновок OC0 поводиться однаково, а ось в режимах широтно-імпульсної модуляції його поведінка відрізняється.

І останній біт регістра TCCR0 - це біт **FOC0 (Force Output Compare)**. Цей біт призначений для примусового зміни стану виведення OC0. Він працює тільки для режимів Normal і CTC. При установки біта FOC0 в одиницю стан виведення змінюється відповідно значенням бітів COM01, COM00. FOC0 біт не викликає переривання і не скидає таймер в CTC режимі.

TIMSK (Timer / Counter Interrupt Mask Register)

WGM01	WGM00	Режим работы таймера/счетчика
0	0	Normal
0	1	PWM, Phase Correct
1	0	CTC
1	1	Fast PWM

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
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	

Загальний реєстр для всіх трьох таймерів ATmega16, він містить прапори дозволу переривань. Таймер T0 може викликати переривання при переповненні рахункового регістра TCNT0 і при збігу рахункового регістра з регістром порівняння OCR0. Відповідно для таймера T0 в регістрі TIMSK зарезервовані два біта - це TOIE0 і OCIE0. Решта біти відносяться до інших таймерам.

TOIE0 - 0-е значення біта забороняє переривання за подією переповнення, 1 - дозволяє.
OCIE0 - 0-е значення забороняє переривання за подією збіг, а 1 дозволяє.

Природно переривання будуть викликатися, тільки якщо встановлений біт глобального дозволу переривань - біт I регістра SREG.

TIFR (Timer / Counter0 Interrupt Flag Register)

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

Загальний для всіх трьох таймерів-лічильників регістр. Містить статусні прапори, які встановлюються при виникненні подій. Для таймера T0 - це переповнення рахункового регістра TCNT0 і збіг рахункового регістра з регістром порівняння OCR0.

Якщо в ці моменти в регістрі TIMSK дозволені переривання і встановлений біт I, то мікроконтролер викличе відповідний обробник.

Прапори автоматично очищаються при запуску обробника переривання.Також це можна зробити програмно, записавши 1 до відповідного прапор.**TOV0** - встановлюється в 1 при переповненні рахункового регістра. **OCF0** - встановлюється в 1 при збігу рахункового регістра з регістром порівняння

SFIOR (Special Function IO Register)

Bit	7	6	5	4	3	2	1	0	
	ADTS2	ADTS1	ADTS0	–	ACME	PUD	PSR2	PSR10	SFIOR
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Один з його розрядів скидає 10-ти розрядний двійковий лічильник, який ділить вхідну частоту для таймера T0 і таймера T1.

Скидання здійснюється при установці біта **PSR10 (Prescaler Reset Timer / Counter1 і Timer / Counter0)** в одиницю.

Вісьмирозрядні таймери T0, T2

Таймери T0 і T2 є вісьмиразрядними й присутні у всіх моделях ATmega. Рахункові регістри таймерів - TCNTn (тут і далі n - номер лічильника).

Тактирование таймерів здійснюється від основного генератора че- рез предделитель, при цьому коефіцієнт розподілу частоти задається бітами CSn0, CSn1 і CSn2 регістри TCCRn [1, 3].

Для таймера T0:

TCCRn|=(1<<CSn0); //Тактувати без розподілу частоти.

TCCRn|=(1<<CSn1); //Тактувати із коефіцієнтом розподілу 8.

TCCRn|=(1<<CSn0)|(1<<CSn1); //Тактувати із коефіцієнтом 64.

TCCRn|=(1<<CSn2); //Тактувати із коефіцієнтом 256.

TCCRn|=(1<<CSn0)|(1<<CSn2); //Тактувати із коеф. 1024.

TCCRn&=~(1<<CSn0|1<<CSn1|1<<CSn2); //Зупинити лічильник.

Для таймера T2:

TCCRn|=(1<<CSn0); //Тактувати без розподілу частоти.

TCCRn|=(1<<CSn1); //Тактувати із коефіцієнтом розподілу 8.

TCCRn|=(1<<CSn0)|(1<<CSn1); //Тактувати із коефіцієнтом 32.

TCCRn|=(1<<CSn2); //Тактувати із коефіцієнтом 64.

TCCRn|=(1<<CSn0)|(1<<CSn2); //Тактувати із Коеф. 128.

TCCRn|=(1<<CSn1)|(1<<CSn2);

//Тактувати із Коеф. 256. TCCRn|=(1<<CSn0)|(1<<CSn1)|(1<<CSn2);

//Дільник 1024.

TCCRn&=~(1<<CSn0|1<<CSn1|1<<CSn2); //Зупинити лічильник.

Вісьмиразрядні таймери можуть працювати в наступних режимах:

- переривання по переповненню;
- переривання по збігу з регістром порівняння;
- зміна стану виведення OCn при збігу з регістром порівняння;
- асинхронний режим тактирования;
- режим Fast PWM.

Переривання по переповненню [1].

При переході значення лічильника з максимального в нульове встановлюється прапор переповнення TOVn. Якщо при цьому встановлений прапор дозволу переривання по переповненню TOIEn регістра TIMSK і пере-

ривання дозволені глобально, то відбудеться виклик оброблювача перерывания по переповненню.

Переривання по збігу з регістром порівняння [3].

Всі таймери, крім T0, здатні формувати переривання при збігу поточного значення лічильника TCNTn із умістом регістра OCRn. Порівняння зазначених регістрів відбувається в кожному машинному циклі. У випадку їхньої рівності встановлюється прапор OCFn регістра TIFR і генерується відповідне переривання, якщо воно дозволено установкою біта OCIEn того ж регістра:

Зміна стану виведення OCn при збігу з регістром порівняння

Момент збігу вмісту регістрів лічильника й порівняння можна використати для керування станом зовнішнього виведення OC0(OC2). Для цього необхідно:

1) установити висновок OCn у стан виходу;

2) дозволити даний режим установкою біта FOCn регістра TCCRn;

3) визначити дії над висновком комбінацією битов COMn0 і COMn1 у регістрі TCCRn:

- COMn1=0, COMn0=0 - таймер відключений від виведення OCn;
- COMn1=0, COMn0=1 – стан виведення міняється на противопо-помилкове;
- COMn1=1, COMn0=0 - висновок скидається в нуль; COMn1=1,
- COMn0=1 - висновок установлюється в одиницю.

Помітимо, що в даному режимі виклик переривання блокується. Це утрудняє скидання лічильника після збігу й при генеруванні зовнішнього сигналу із заданою частотою. Вирішити цю проблему можна, дозволивши режим CTC (скидання при збігу) установкою біт WGMn1 в TCCRn. При цьому лічильник після збігу з регістром порівняння буде скидатися автоматично.

Приклад: Генерувати частоту 1кгц на виведення OC2 ATmega з генератором на 1 МГц

```
OCR2=(125-1); //При частоті 1 Мгц і дільнику 8

                    //для 1ms - 125 тактів таймера.

TCCR2|=(1<<CS01); //Тактувати із коефіцієнтом розподілу 8.

DDRB|=(1<<3);//Установити висновок OC2 (Port.3) на вихід

//Дозволити зміна виведення OC2. Включити режим CTC.

TCCR2|=(1<<FOC2)|(1<<COM20)|(1<<WGM21);
```

Асинхронний режим тактирования

Таймер T2 має можливість тактирования від власного асинхронного генератора з висновками для підключення зовнішнього кварцового резонатора - TOSC1 і TOSC2. Перемикання в асинхронний режим виробляється

установкою біта ASn у регістрі ASSR. Підключивши до вказанних виводів кварц 32768 Гц, можна організувати підрахунок часу з початку роботи програми:

```
#define LED 0              //Висновок світлодіода.

char Sec,Min,Hour; //Лічильники секунд, мінут, часов. void
main (void)

{

 DDRA|=(1<<LED);    //Світлодіод на вихід
 ASSR|=(1<<AS2);     //Тактувати T2 від асинхронного

                     //генератора таймера

//Тактувати частотою 32768/1024=32 Гц,

//режим скидання по збігу.
TCCR2|=(1<<CS20)|(1<<CS21)|(1<<CS22)|(1<<WGM21);

OCR2=(32-1); //(для 32 тактів OCR0=31!),

            //збігу із частотою 1 Гц.

TIMSK|=(1<<OCIE2); //Дозволити переривання по збігу T2. SREG |=
(1<<7);             //Дозволити переривання.

while (1) {}

}

//Оброблювач щосекундних переривань T2.

#pragma vector=TIMER2_COMP_vect

_interrupt void T2_COMP()

{

PORTA^=(1<<LED); //Нова секунда. if
(++Sec==60)

        {

        Sec=0; //Нова хвилина.

        if (++Min==60)

                {Min=0;Hour++;} //Нова година.

        }
```

}

Шістнадцятирозрядний таймер Т1 [1, 3]

Шістнадцятирозрядний таймер-лічильник Т1 входить до складу всіх моделей мікроконтролерів серії Mega. Рахунковий регістр таймера

TCNT1 = TCNT1H*256 + TCNT1L,

де TCNT1H - старший байт, TCNT1L - молодший байт.

За замовчуванням таймер тактируется від основного генератора через предделитель, що знижує частоту зміни значень рахункового реги- стра в 8, 64, 256 або 1024 рази залежно від комбінації битов CS10, CS11 і CS12 у регістрі TCCR1B:

```
TCCRnB|=(1<<CSn0); //Тактувати прямо від генератора.
TCCRnB|=(1<<CSn1); //Тактувати із коефіцієнтом розподілу 8.
TCCRnB|=(1<<CSn0)|(1<<CSn1); //Тактувати із Коеф. 64.

TCCRnB|=(1<<CSn2); //Тактувати із коефіцієнтом 256.
TCCRnB|=(1<<CSn0)|(1<<CSn2); //Тактувати із Коефф. 1024.
TCCRn&=~(1<<CSn0|1<<CSn1|1<<CSn2); //Зупинити лічильник.
```

Шістнадцятирозрядний таймер може працювати в режимах: переривання

- по переповненню;
- переривання по збігу з регістром порівняння; режим
- CTC;
- зміна стану виведення OC1A при збігу з регістром порівняння.

Переривання по переповненню

Класичний варіант - переривання по переповненню рахункового регістра викликається при переході значення рахункового регістра з максимального (0xFFFF) у нульове. При цьому встановлюється прапор TOV1 регістра TIFR, що скидаються апаратно при вході в оброблювач переривання. Дозволяється переривання установкою біта TOIE11 у регістрі TIMSK.

Переривання по збігу з регістром порівняння

До складу таймера Т1 входять шестнадцатиразрядные регістри сравнения OCR1A, OCR1B і OCR1C, уміст яких у кожному машинному циклі рівняється з поточним значенням рахункового регістра TCNT1.

У випадку їхнього збігу встановлюються прапори OCF1A, OCF1B і OCF1C. Якщо відповідне переривання дозволене установкою битів OCIE1A і OCIE1B регістра TIMSK, то відбувається виклик обробника переривання від регістра порівняння. Прапори збігу, як і прапори переповнення, при вході в оброблювач скидаються апаратно [1, 3].

Режим CTC (скидання при збігу)

Шістнадцятирозрядний таймер Т1 має можливість самостійно тельно скидати рахунковий регістр відразу після його збігу з регістром порівняння, після чого рахунок автоматично триває з нульового значення. Такий режим

називається режимом скидання при співпадінні (CTC) і може бути використаний, зокрема, для генерації сигналів фіксованої частоти. Для перекладу таймера в даний режим необхідно встановити біти WGM12 і WGM13 регістра TCCR1B.

Зміна стану виведення OC1A при збігу з регістром порівняння

У всіх мікроконтролерах ATmega є вивід з альтернативною функцією OC1A. Настроєний на вихід, він може змінювати свій стан у момент збігу значень рахункового регістра й регістра порівняння. Характер цих змін визначається комбінацією битів COM1A0 і COM1A1 у регістрі TCCR1A, наприклад:

- COM1A1=0, COM1A0=0 - таймер відключений від виводу OC1A;
- COM1A1=0, COM1A0=1 - стан виводу міняється на обернений;
- COM1A1=1, COM1A0=0 - OC1A скидається в нуль; COM1A1=1, COM1A0=1 - OC1A встановлюється в одиницю.

Дозволяється керування виводом установкою біта FOC1A в TCCR1C, однак при цьому блокується виклик переривання по збігу.

Примітки: 1. Для забезпечення одночасного запису обох байтів у шістнадцятирозрядні регістри AVR старший байт попередньо міститься у часовий регістр TEMP. Потім при записі молодшого байта відбувається їх одночасне переміщення в шістьнадцятирозрядний регістр. Для того, щоб виключити запис у старший байт випадкового значення, що зберігається в буфері TEMP, запис молодшого байта необхідно робити після запису старшого.

2. Не слід забувати, що ініціалізаційні значення, наприклад для регістра порівняння, повинні бути на одиницю менше числа тактів у заданому інтервалі часу плюс один такт для переходу від максимального значення до нульового.

Лекція 5. Поняття ШІМ. АЦП та компаратор.

Питання:

1.Поняття ШІМ.

2. Функціонування АЦП в мікроконтролерах АЦП.

3. Аналоговий компаратор в мікроконтролерах АЦП..

1.Поняття ШІМ.

За переривання від таймерів відповідають регістри TIMSK, TIFR. TIMSK - це регістр масок. Тобто біти, що знаходяться в ньому, локально дозволяють переривання. Якщо біт встановлено, значить, конкретне переривання дозволено. Якщо біт в нулі, значить, дане переривання не обробляється. За замовчуванням всі біти в нулі. За переривання за переповненням відповідають біти:

- TOIE - дозвіл на переривання по переповненню таймера 0
- TOIE1 - дозвіл на переривання по переповненню таймера 1
- TOIE2 - дозвіл на переривання по переповненню таймера 2

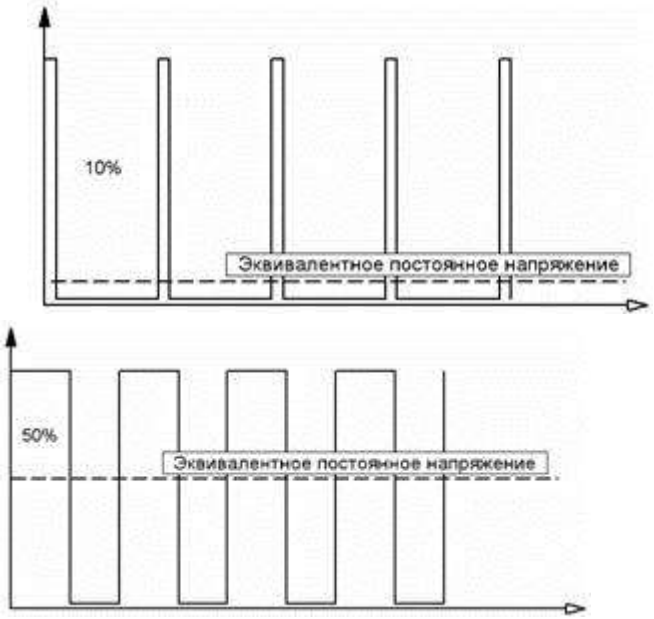
Регістр TIFR це безпосередньо флагової регістр. Коли якийсь переривання спрацьовує, то прапор переривання встановлюється. Цей прапор скидається апаратно, коли програма йде по вектору. Якщо переривання заборонені, то прапор так і буде стояти до тих пір, поки переривання не дозволять і програма не втече на переривання. Щоб цього не сталося, прапор можна скинути вручну. Для цього в TIFR в нього потрібно записати 1

Широтно Імпульсна Модуляція

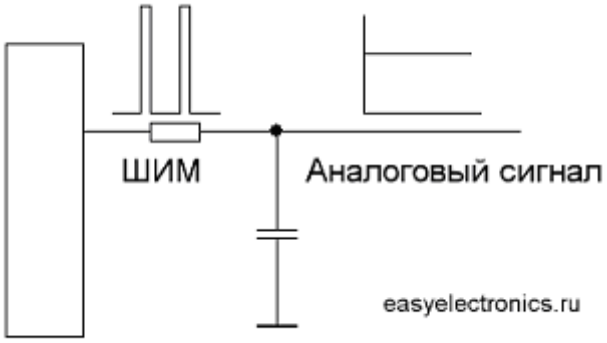
Широтно-імпульсна модуляція **(** PWM - Pulse Width Modulation **)** це спосіб завдання аналогового сигналу цифровим методом **,** тобто з цифрового виходу, що дає тільки нулі і одиниці отримати якісь плавно змінюються величини.

Уяви собі важкий маховик, який ти можеш обертати двигуном. Причому двигун ти можеш або включити, або вимкнути. Якщо включити його постійно, то маховик розкрутиться до максимального значення, і так і буде крутитися. Якщо вимкнути, то зупиниться за рахунок сил тертя. А ось якщо двигун включати на десять секунд кожну хвилину, то маховик розкрутиться, але далеко не на повну швидкість - велика інерція згладить ривки від включається двигуна, а опір від тертя не дасть йому крутитися нескінченно довго. Чим більше тривалість включення двигуна в хвилину, тим швидше буде крутитися маховик.

При ШІМ ми подаємо на вихід сигнал, що складається з високих і низьких рівнів (застосовно до нашої аналогії - включаємо і вимикаємо двигун), тобто нулів і одиниць. А потім це все пропускається через інтегруючу ланцюжок (в аналогії - маховик). В результаті інтегрування на виході буде величина напруги, що дорівнює площі під імпульсами.Змінюючи **шпаруватість** (відношення тривалості періоду до тривалості імпульсу) можна плавно змінювати цю площу, а значить і напруга на виході. Таким чином, якщо на виході суцільні 1, то на виході буде напруга високого рівня (наприклад, 12 вольт), якщо нулі, то нуль. А якщо 50% часу буде високий рівень, а 50% низький то 6 вольт. Інтегрує ланцюжком тут буде служити маса якоря двигуна, що володіє досить велику інерцію.



А що буде, якщо взяти і дати ШІМ - сигнал не з нуля до максимуму, а від мінуса до плюса. Скажімо від +12 до -12. А можна задавати змінний сигнал! Коли на вході нуль, то на виході 12В, коли один, то +12. Якщо шпаруватість 50% то на виході 0В. Якщо шпаруватість міняти за синусоїдальним законом від максимуму до мінімуму, то отримаємо змінну напругу.А якщо взяти три таких ШІМ генератора і гнати через них синусоїди, зрушені на 120 градусів між собою, то отримаємо звичайнісіньке трифазну напругу, а значить привіт безколекторні асинхронні і синхронні двигуни **.** На цьому принципі побудовані всі сучасні промислові приводи змінного струму. Як згладжує інтегруючого ланцюга в ШІМ може бути застосована RC ланцюжок:



Апаратна реалізація ШІМ

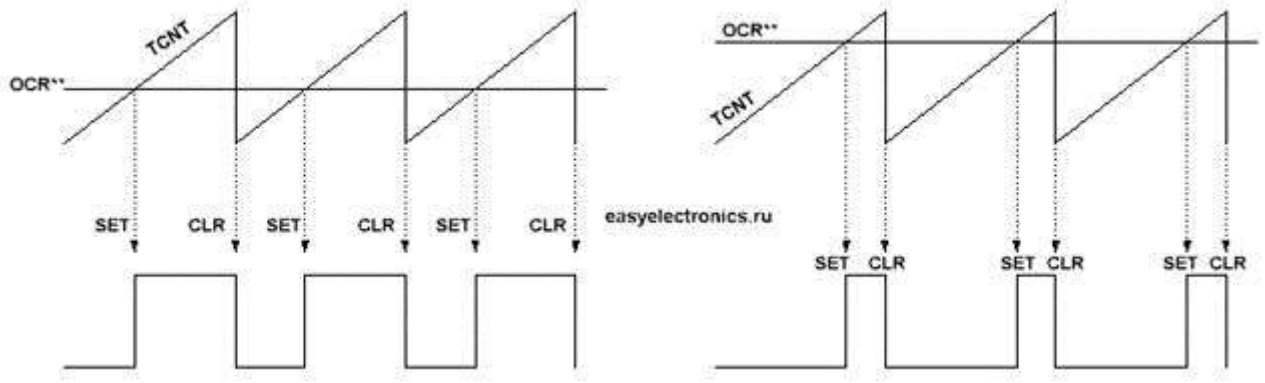
У разі **ATMega** найпростіше зробити на його ШІМ-генераторі, який вбудований в таймери. Причому в першому таймері є два канали. Так що ATmega може реалізувати одночасно чотири канали ШІМ **.**

У таймера є особливий регістр порівняння **OCR .** Коли значення в рахунковому регістрі таймера досягає значення знаходиться в регістрі порівняння, то можуть виникнути наступні апаратні події:

- Переривання за випадковим збігом
- Зміна стану зовнішнього виходу порівняння **OC .**

Виходи порівняння виведені назовні, на висновки мікроконтролера. Припустимо, що ШІМ - генератор налаштований так , що коли значення в рахунковому регістрі більше , ніж в регістрі порівняння, то на виході 1, а коли менше, то 0. Що при цьому відбудеться? Таймер буде вважати, як йому і належить, від нуля до 256 з частотою , яку ми налаштуємо битами

предделителя таймера. Після переповнення скидається в 0 і продовжує рахувати заново.



На виході з'являються імпульси. Якщо збільшити значення в регістрі порівняння, то ширина імпульсів стане вужче. Тобто, змінюючи значення в регістрі порівняння , можна змінювати шпаруватість ШІМ - сигналу. А якщо пропустити цей ШІМ-сигнал через згладжує RC ланцюжок (інтегратор) , то отримаємо аналоговий сигнал.

У таймера може бути декілька регістрів порівняння. Залежить від моделі МК і типу таймера. У нових AVR буває і по три регістра порівняння на таймер, що дозволяє одним МК організувати декілька незалежних ШІМ каналів. Самих режимів ШІМ існує декілька:

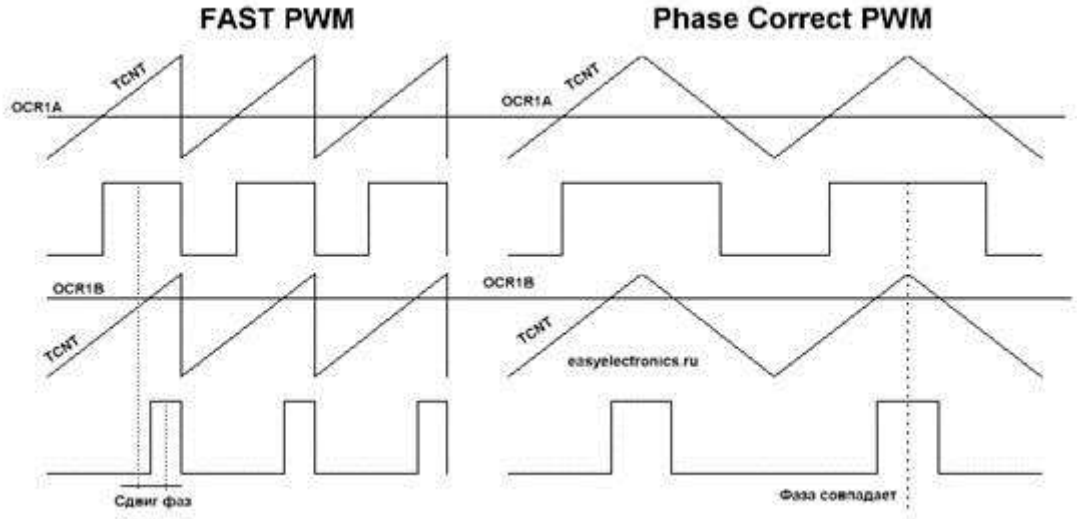
Швидкий ШІМ (Fast PWM)

В цьому режимі лічильник рахує від нуля до 255 , після досягнення переповнення скидається в нуль і рахунок починається знову. Коли значення в лічильнику досягає значення регістра порівняння, то відповідний йому висновок **OCxx** скидається в нуль. При обнулення лічильника цей висновок встановлюється в 1.

Частота отриманого ШІМ сигналу визначається просто: частота процесора, наприклад, 8МГц, таймер цокає до 256 з тактовою частотою. Значить, один період ШІМ буде дорівнює 8000 000/256 = 31250Гц. Швидше не вийде - це максимальна швидкість на внутрішньому 8МГц тактовом генераторі. Ще є можливість підвищити дозвіл, зробивши рахунок 8, 9, 10 розрядним (якщо розрядність таймера дозволяє), але треба враховувати, що підвищення розрядності, разом з підвищенням дискретності вихідного аналогового сигналу, різко знижує частоту ШІМ.

ШІМ з фазовою корекцією (Phase Correct PWM)

ШІМ з точною фазою. Працює схоже, але тут лічильник рахує дещо по-іншому. Спочатку від 0 до 255, потім від 255 до 0. Висновок OCxx при першому збігу скидається, при другому встановлюється. Але частота ШІМ при цьому падає вдвічі, через більшого періоду. Основне його призначення, робити багатофазні ШІМ сигнали, наприклад, трифазну синусоїду. Тобто центри імпульсів в різних каналах і на різній скважности будуть збігатися.



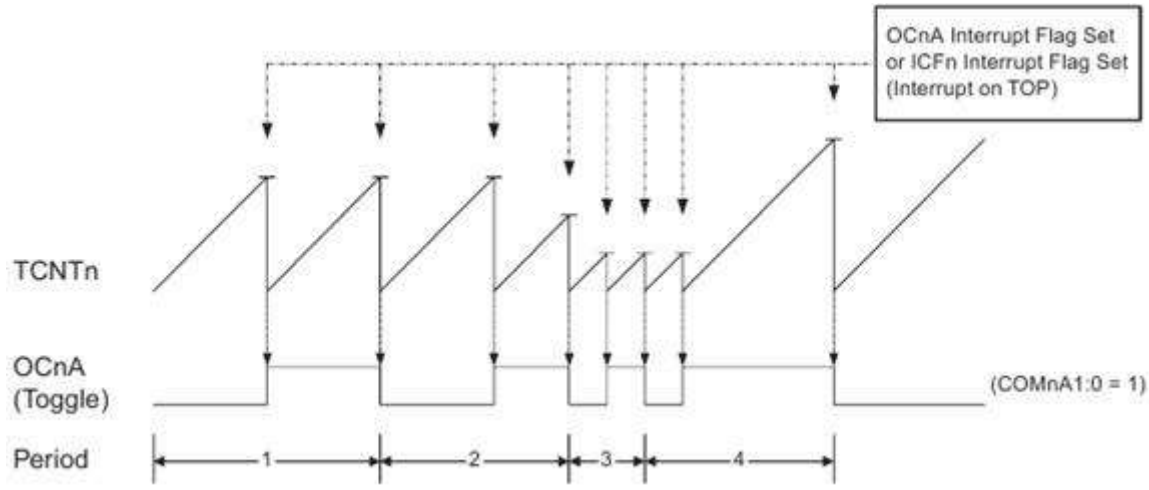
Мал. Режим швидкої ШІМ і ШІМ з фазовою корекцією

Щоб не було зайвих імпульсів, в регістр порівняння будь-яке значення потрапляє через буферний регістр і заноситься тільки тоді, коли значення в лічильнику досягне максимуму. Тобто до початку нового періоду шим імпульсу.

Скидання по збігу (Clear Timer On Compare)

Скидання при порівнянні. ЧІМ - частотно-імпульсно модельованої сигнал. Тут працює трохи інакше, ніж при інших режимах. Тут рахунковий таймер цокає немає від 0 до межі, а від 0 до регістру порівняння! А після чого скидається.

В результаті, на виході виходять імпульси завжди однаковою скважности, але різної частоти. А найчастіше цей режим застосовується, коли треба таймером відраховувати періоди (і генерувати переривання) із заданою точністю.



Наприклад, треба нам переривання кожну мілісекунду. І щоб ось точно. Як це реалізувати простіше? Через Режим CTC! Нехай частота дорівнює 8МГц. Переддільник буде дорівнює 64, таким чином, частота рахунку таймера складе 125000 Гц. А нам треба переривання з частотою 1000Гц. Тому налаштовуємо переривання за випадковим збігом з числом 125. Дійшов рахунок до 125 - виробилося переривання, обнулився лічильник. Дійшов до 125 - дав переривання, обнулився.

Режим Fast PWM - швидкодіючий ШІМ

Разом з висновками OC0/OC2, таймери T0/T2 можуть використовуватися для генерування сигналу із широтно-імпульсною модуляцією. Такий сигнал характеризується постійною частотою й шпаруватістю, що змінюється (коефіцієнтом заповнення) імпульсів

(співвідношенням тривалості одиничного й нульового рівнів сигналу). У режимі Fast PWM відбувається автоматична установка в одиницю виведення OCn при переповненні лічильника і його скидання при збігу лічильника з регістром порівняння OCRn [1, 3].

Як бачимо, частота сигналу ШІМ може регулюватися тільки частотою генератора й підбором коефіцієнта дільника таймера й не може бути вище 62,5 кгц (16Мгц / 256).

Для перекладу таймера в режим «Швидкодіючий ШІМ» треба вибрати його установкою битів WGMn0 і WGMn1 у регістрі TCCRn:

TCCRn|=(1<<WGMn0)|(1<<WGMn1);

Функція виведення OCn при цьому задається в тім же регістрі бітами COMn0 і COMn1:

COMn1=0, COMn0=0 - таймер відключений від виведення OCn;

COMn1=0, COMn0=1 - зарезервована комбінація;

COMn1=1, COMn0=0 - нормальний ШІМ;

/ COMn1=1, COMn0=1 - інвертований ШІМ (скидання виходу при пе- реповнені таймера й установка при збігу з регістром порівняння).

У даному режимі доступне як переривання по переповненню, так і по збігу з регістром порівняння, в оброблювачах яких можна регулювати шпаруватість вихідного сигналу від 0 до 100 % шляхом зміни вмісту регістра порівняння OCRn. При цьому скважність дорівнює OCRn/256.

2. **Функціонування АЦП в мікроконтролерах АЦП.**

ATMega16 містить в собі 10-бітовий АЦП, вхід якого може бути з'єднаний з одним з восьми висновків Port A. АЦП Mega16, як і будь-якого іншого АЦП, потрібно опорна напруга для цілей порівняння з вхідним (якщо вимірюється одно опорного, то отримуємо максимальний код в двійковому вигляді). Опорна напруга подається на висновок ADRef або може використовуватися внутрішній генератор з фіксованою напругою 2,65 В. Отриманий результат можна уявити в такому вигляді:

$$ADC=\frac{Vin*1024}{Vref}$$

АЦП включається установкою біта ADEN в регістрі ADCSRA. Після перетворення, 10-бітний результат виявляється в 8-бітних регістрах ADCL і ADCH. За замовчуванням, молодший біт результату знаходиться праворуч (тобто в bit 0 регістра ADCL, так зване праве орієнтування). Але порядок проходження бітів на ліве орієнтування можна змінити встановивши біт ADLAR в регістрі ADMUX. Це зручно, якщо потрібно отримати 8-бітовий результат. У такому випадку потрібно прочитати тільки регістр ADCH. В іншому випадку, Ви повинні спочатку прочитати регістр ADCL першим, а ADCH другим, щоб бути впевненим в тому, що читання цих двох регістрів відноситься до результату одного перетворення.

Одиночне перетворення може бути викликано записом біта ADSC в регістр ADCSRA. Цей біт залишається встановленим весь час, займанийперетворенням. Коли перетворення закінчено, біт автоматично встановлюється в 0. Можна також починати перетворення щодо подій з різних джерел. Модуль АЦП також може працювати в режимі "вільного польоту". В такому випадку АЦП постійно виробляє перетворення і оновлює регістри ADCH і ADCL новими значеннями.

Ви зможете конвертувати модулю АЦП необхідна тактова частота. Чим вище ця частота, тим швидше буде відбуватися перетворення (воно, звичайно, займає 13 тактів, перше перетворення займає 25 тактів). Але чим вище частота (і вище швидкість перетворення), тим менш точним виходить результат. Для отримання максимально точного

результату, модуль АЦП повинен тактуватися частотою в межах від 50 до 200 КГц. Якщо необхідний результат з точністю менше 10 біт, то можна використовувати частоту більше 200 КГц. Модуль АЦП містить дільник частоти, щоб отримувати потрібну тактову частоту для перетворення з частоти процесора.

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
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	

Регістр **ADMUX** задає вхідний контакт порту А для підключення АЦП, орієнтування результату і вибір опорної частоти. Якщо встановлений біт ADLAR, то результат ліво-орієнтований. Опорна частота від внутрішнього генератора задається виставленими в 1 битами REFS1 і REFS0. Якщо обидва біта скинуті, то опорна частота береться від контакту AREF. У разі, якщо REFS1 = 0 а REFS0 = 1, опорна частота береться від AVCC із зовнішнім конденсатором, підключеним до AREF. Вибір контакту введення виконується наступним чином:

MUX4..0	Single Ended Input
00000	ADC0
00001	ADC1
00010	ADC2
00011	ADC3
00100	ADC4
00101	ADC5
00110	ADC6
00111	ADC7

Регістр контролю та статусу АЦП **ADCSRA** :

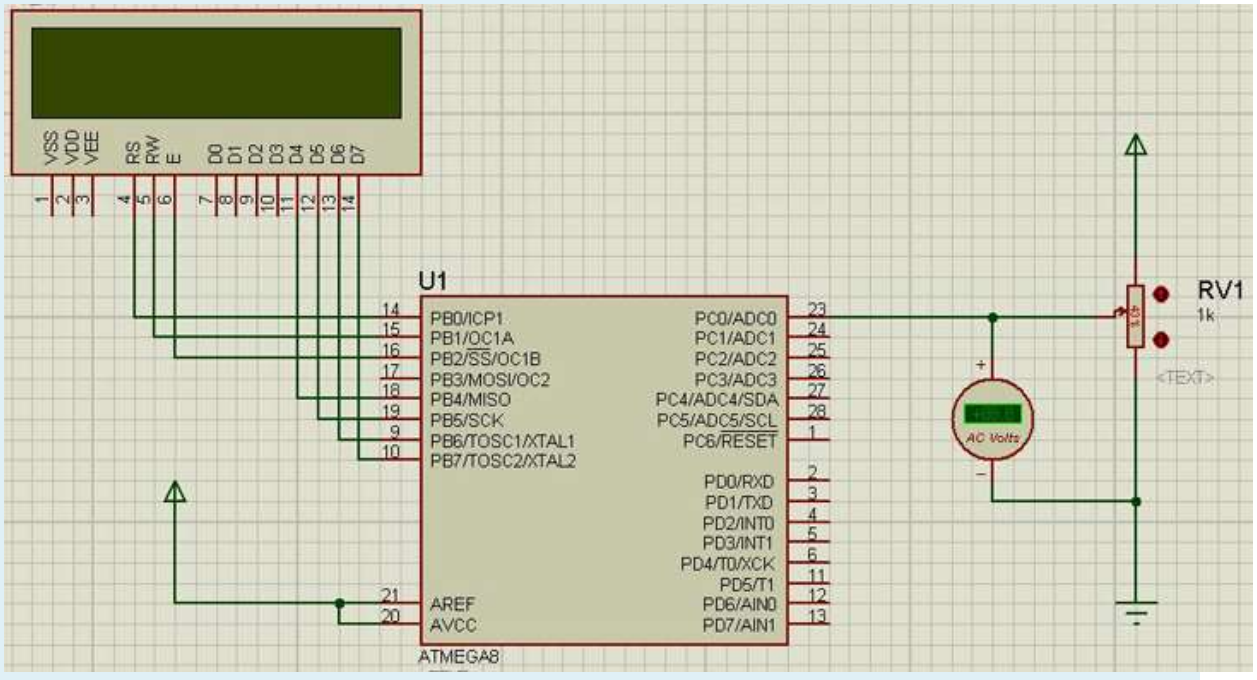
Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Біт ADEN = 1 включає модуль АЦП.
Запис одиниці в ADSC запускає цикл перетворення. У режимі "вільного польоту" запис одиниці запускає перше перетворення, наступні запускаються автоматично.
ADIF - прапор переривання АЦП. Цей біт встановлюється в 1 коли АЦП завершено перетворення і в регістрах ADCL і ADCH знаходяться актуальні дані. Цей прапор встановлюється навіть в тому випадку, якщо переривання заборонені. Це необхідно для випадку програмного опитування АЦП. Якщо використовуються переривання, то прапор скидається автоматично. Якщо використовується програмний опитування, то прапор може бути скинутий записом лог.1 в цей біт.
ADIE - Якщо в цьому бите встановлена одиниця, і переривання дозволені глобально, то при закінченні перетворення буде виконано перехід по вектору переривання від АЦП.
Біти ADPS2..0 задають коефіцієнти предделителя частоти:

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

МК Atmega8. Даний МК має 6 каналів 10-ти розрядних АЦП, два з яких 8-ми розрядні. Для того щоб почати працювати з АЦП, нам знадобитися лише опорна напруга. Для початку візьмемо 5 вольт. Це означає що отціфровка сигналу можлива від 0 до 5 вольт. Так як канал 10-ти розрядний то виходить $5/2^{10} = 0.0048$ В на одиницю. Значить якщо у нас АЦП видасть при вибірці число 482, то напруга буде дорівнювати 482 * 0.0048 = 2.3136 В. В принципі непогана точність. За заявленим параметрам, МК робить 15000 вибірок в секунду. Звідси наш графік можна розбити на 14999 прямих.

Малюнок 8.



На порт У МК приєднаний LCD 16x2. . Висновки AREF і AVCC приєднані до 5В. Це є той опорна напруга. На порт С до нульового розряду приєднаний контакт з вольтметром і змінним резистором для зміни вхідного напруги. Наше завдання така.

Ми повинні вивести на екран величину напруги ту яку показує вольтметр.

Запускаємо новий проект. В налаштуваннях вибираємо Atmega8, 4,00000000 MHz. У вкладці LCD вибираємо PORTB.

Насамперед ми додаємо дві директиви препроцесора для роботи з текстом і затримки. Для цього після директиви LCD додамо
ці два рядки.

```
#include <delay.h>
#include <stdio.h>
```

Перша з них потрібна для створення затримок, а друга для роботи з текстом. далі нам потрібно створити масив для тимчасового зберігання тексту фіксованої. після написи "// Declare your global variables here" пишемо char string [10] ;.

```
// Declare your global variables here
char string [10];
```

Тепер після відкриття головної функції main, ми повинні оголосить дві змінні.
Одна для зберігання значення після вибірки, а інша для зберігання виведеного значення.
Для цього запишемо так.

```
void main (void)
{
// Declare your local variables here
int data; // Змінна для зберігання даних вибірки. int так як регістр 10 розрядів.
float V;   // Змінна для виведеного значення. float так як у нас точність до 2 знаків.
```

Тепер давайте займемося налаштуванням самого АЦП. Для цього після настройка компаратора відразу запишемо два рядки.

```
// Analog Comparator initialization
// Analog Comparator: Off
// Analog Comparator Input Capture by Timer / Counter 1: Off
ACSR = 0x80;
SFIOR = 0x00;
```

```
ADMUX = 0;      // Перший рядок, № порту.
ADCSR = 0x85;   // Другий рядок настройка АЦП.
```

Ну з номером порту все ясно, який номер прописаний з тим і працюємо, а ось з налаштуванням
тут давайте по докладніше. Для того щоб почати роботу з АЦП у МК є такий регістр який називається **ADCSR** . Що в ньому знаходиться.

0-й біт **ADPS0** Вибір частоти перетворення
1-й біт **ADPS1** Вибір частоти перетворення

2-й біт **ADPS2** Вибір частоти перетворення
3-й біт **ADIE** Дозвіл переривання
4-й біт **ADIF** Прапор переривання
5-й біт **ADFR** Вибір роботи АЦП. 1-безперервний 0-за запуску ADSC
6-й біт **ADSC** Запуск перетворення 1-старт. після перетворення
скидається в нуль апаратно.
7-й біт **ADEN** Дозвіл роботи АЦП 1-й 0-немає

Тепер давайте його налаштуємо. Щоб включити АЦП нам треба виставити в 1 7-й розряд.

Далі виставимо 0 для старту перетворення.

в 0 5-й розряд. Будемо самі запускати перетворення.

3-й і 4-й розряди виставимо в 0. Ми не будемо працювати з перериванням. Тепер залишилося підібрати частоту.

Якщо почитати мануал на МК, то там сказано що для більш стабільної роботи АЦП його необхідно тактіровать частотою в межах 50 кГц - 200 кГц. Так як у нас кварц на 4000 кГц, то нам його треба поділити. Ось для цього ми і скористаємося першими трьома розрядами. Дивимося нижче на таблицю коефіцієнтів розподілу.

ADPS2	ADPS1	ADPS0	дільник
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Що нам вибрати. Беремо нашу частоту кварцу і ділимо на коефіцієнт.
4000 кГц ділимо на 8, отримуємо 500 кГц. Багато. Давайте тепер на 16, отримаємо 250 кГц, теж забагато. Ділимо на 32 і отримуємо 125 кГц. У те, що треба. ми вклалися в задані межі. Дивимося в таблицю і бачимо що коефіцієнт 32 задається значенням розрядів 101. Ну ніби все зібрали. тепер давайте подивимося на наш регістр. Ось його значення. 10000101 тут зібрані всі раніше розглянуті значення всіх розрядів. Якщо перевести це число в HEX то отримаємо 0x85.
Тепер вам зрозуміло чому я записав в регістр **ADCSR** значення 0x85.
Для цього після ініціалізації LCD запишемо рядок. lcd_putsf ("Work with ADC");

```
// LCD module initialization
lcd_init (16);
lcd_putsf ( "Work with ADC"); // Виводимо запис
```

Тепер при старті програми ми в першому рядку побачимо нашу напис. Далі в нескінченному
циклі пишемо тіло самої програми.

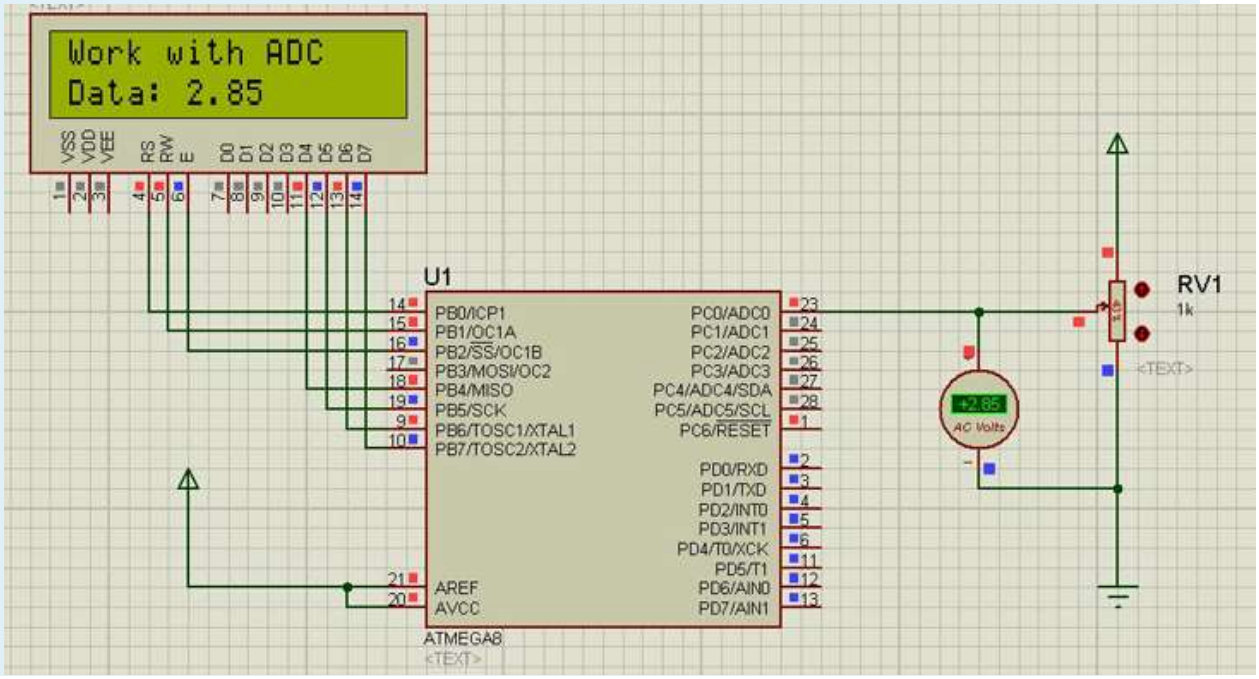
```
while (1)
    {
```

```
        delay_ms (20); // Задаємо затримку в 20 мілісекунд
        ADCSR | = 0x40; // Записуємо 1 в ADSC
        data = ADCW; // вичитувати значення
        V = (float) data * 0.0048828; // Переводимо в вольти
        sprintf (string, " Data:% 1.2f ", V); // форматіруем
        lcd_gotoxy (0,1); // Виставляємо курсор
        lcd_puts (string); // Виводимо значення

      };
```

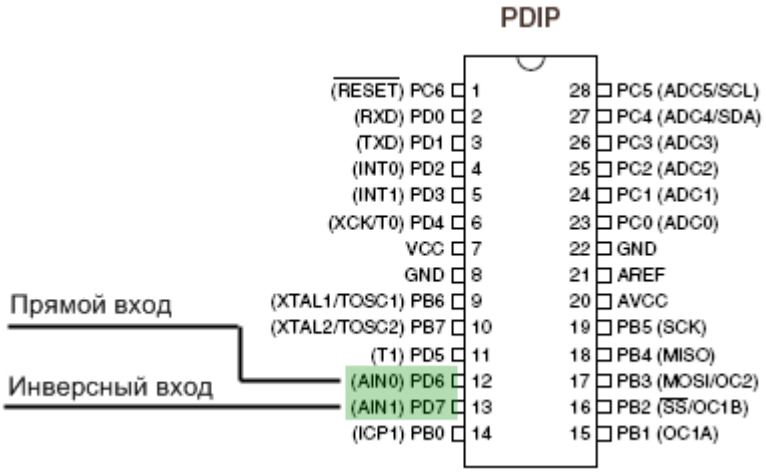
delay_ms (20); Ця функція яка створить затримку на 20 мілісекунд.
ADCSR | = 0x40; Тут ми робимо по бітове АБО. Число 0x40 в бінарі виглядає так 0b01000000. Якщо ми проведемо по бітове АБО з 0x85 (0b10000101), то у нас в 6-й розряд запишеться 1. Пам'ятайте, що треба зробити щоб почалося перетворення. Так, так, саме в 6-й розряд потрібно записати 1. А після перетворення він скинеться в 0 апаратно.
data = ADCW; Після перетворення МК записує отримане значення в регістр ADCW. Ось ми звідти його і висмикуємо.
V = (float) data * 0.0048828; Тут ми перетворимо отримане число в вольти. так як у нас опорна напруга 5В, а значення регістра 1024, то ми 5/1024 = 0.0048828 Це коефіцієнт напруги. Ну або мінімальна величина напруги при мінімальному значенні регістра ADCW. Тобто якщо в регістрі буде значення 1, то велечіна напруги буде дорівнює 0.0048828 В. Тому ми в рядку, дані ADCW перемножуємо з 0.0048828. Слово float в дужки потрібно для того щоб перетворити змінну data з целочисленной в речову з плаваючою точкою.
sprintf (string, "Data:% 1.2f", V); Тут ми заносимо значення напруги в масив string з форматуванням. Спочатку ми впишемо Data:. Після ставиться знак відсотка. Він говорить про те скільки знаків буде виведено. 1.2f говорить про те що ми хочемо вивести один знак до коми і 2 знака після, а буква f каже що ми маємо справу зі значенням речовим з плаваючою точкою.
lcd_gotoxy (0,1); Виставляємо курсор в нульову позицію у другому рядку.
Lcd_puts (string); Виводимо значення на екран.

Малюнок 9.



3. Аналоговий компаратор в мікроконтролерах АЦП

Він робить він приблизно наступне: є два входи (прямий - **AIN0** , інверсний - **AIN1**), на які подається напруга і якщо напруга на вході **AIN0** більше ніж на **AIN1** він видає на виході 1, інакше 0 (виходом служить прапор регістра компаратора). На**окрему ніжку МК вихід компаратора не виведений** , тільки входи. Компаратор мікроконтролерів розберемо на прикладі ATmega8.



У цьому МК, крім визначення яке з напруг на входах більше, вихід компаратора може підключатися до схеми захоплення Таймера / Лічильника1. Крім того, компаратор має свої переривання, умови спрацьовування яких може налаштовувати користувач - на виході фронт зростає, падає або перемикається в протилежний стан. І ще будь-який з входів АЦП мікроконтролера може бути включений на AIN1 компаратора. Щоб компаратор знаходився в робочому стані входи компаратора повинні бути включені як входи (DDR = 0, PORT = 0) Отже, розберемо які регістри є в МК для настройки компаратора.

ACSR

Bit	7	6	5	4	3	2	1	0	
	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0	ACSR
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	N/A	0	0	0	0	0	

Біт 7 - ACD: біт виключення компаратора. Якщо цей біт встановлений в "1" - компаратор вимкнений. Перед вимиканням компаратора необхідно заборонити виникаючі від нього переривання, інакше при виключенні може виникнути переривання.
Біт 6 - ACBG: вибір опорного напруги аналогового компаратора. Якщо біт встановлено, то на ніжку AIN0 подається фіксований опорна напруга від внутрішнього джерела (1.23 В).
Біт 5 - ACO: вихід компаратора. Якщо ACO дорівнює "1" - напруга AIN0> AIN1, якщо - "0" AIN0 <AIN1.
Біт 4 - ACI: прапор переривання компаратора. "1" - переривання було, "0" - переривання не було.
Біт 3 - ACIE: дозвіл переривання від компаратора. "1" - дозволено, "0" - заборонено.
Біт 2 - ACIC: підключення виходу компаратора до схеми захоплення Таймера / Счетчіка1. "1" - підключений, "0" - відключений.
Біт 1: 0 - ACIS1: ACIS0 : настройка умов спрацьовування переривань від компаратора.

Таблица 1. Установка ACIS1/ACIS0

ACIS1	ACIS0	Interrupt Mode
0	0	Прерывание по изменению на выходе
0	1	Резерв
1	0	Прерывание по падающему фронту
1	1	Прерывание по растущему фронту

Ось і все регістри пов'язані з компаратором мікроконтролера. Так, ще можна сказати про бите **ACME** - 3-й біт регістра **SFIOR** (реєстр спеціальних функцій введення виведення) - якщо в цей біт записаний "0" - вхід компаратора AIN1 підключений до ніжки мікроконтролера AIN1, а якщо "1" - тоді AIN1 може бути підключений до будь-якого входу АЦП мікроконтролера (за умови що АЦП вимкнений - біт **ADEN = 0**регістра **ADCSRA**). Вхід вибирається установкою біт **MUX2: MUX0** регістра АЦП **ADMUX** .

Отже, щоб **підключити вхід АЦП до AIN1** :

1. ACME = 1; // Включити мультиплексор аналогового компаратора
2. ADEN = 0; // Вимкнути АЦП
3. MUX2: MUX0; // Налаштувати потрібний вхід АЦП

Все це намальовано в таблиці 2.

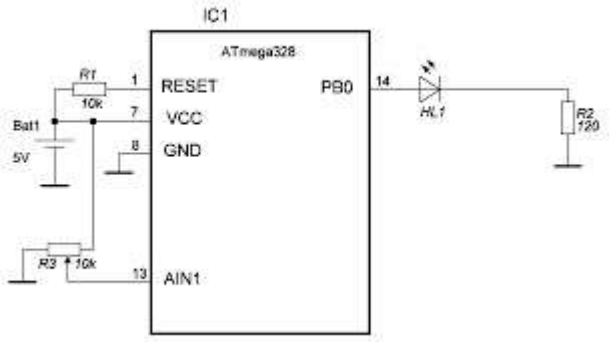
Таблица 2

ACME	ADEN	MUX2:0	Инверсный вход компаратора
0	x	xxx	AIN1
1	1	xxx	AIN1
1	0	000	ADC0
1	0	001	ADC1
1	0	010	ADC2
1	0	011	ADC3
1	0	100	ADC4
1	0	101	ADC5
1	0	110	ADC6
1	0	111	ADC7

Аналоговий компаратор призначений для порівняння рівнів напруг на його входах. Він дуже простий у використанні. Управління здійснюється всього одним регістром, якщо не брати до уваги використання додаткових входів. Аналоговий компаратор має внутрішній ІОН 1.1 В, який ми будемо використовувати в цій програмі.

Якщо до неінвертуючий вхід підключити ІОН, то його напруга також з'явиться на виведення AIN0 (12 ніжка). До висновку AIN1 (13 ніжка) підключимо потенціометр і будемо змінювати напругу на ньому. Якщо напруга на вході AIN1 перевищить 1.1 В, то біт ACO (5) регістра ACSR, що є виходом компаратора, скинеться в 0. Використовуючи цей біт, ми будемо управляти світлодіодом, підключеним до порту B0 (14 ніжка).

Простіше кажучи, якщо напруга на вході AIN1 більше 1.1 В, загоряється світлодіод, якщо менше - гасне.



Малюнок 1 - Схема підключення

Текст програми:

```
#include  <avr / io.h>  // підключення стандартної бібліотеки вводу / виводу

int  main ( void )
{
 // SETUP
 // Comp
 ACSR = 0x40; // підключаємо ІОН до входу AIN0
 // I / O
 DDRB = 0xFF; // порт В як вихід
 // Programm
 while (1)
 {
  while ((ACSR & 0x20) == 0x20) // поки вихід компаратора = 1 (AIN0> AIN1), виконуємо дію
  {
   PORTB = 0x00; // гасимо світлодіод
  }
  PORTB = 0xFF; // запалюємо світлодіод
 }
}
```

При обертанні потенціометра, напруга на вході AIN1 збільшується і стає більше 1.1 В це сприяє запаленню світлодіода. Якщо обертати потенціометр в іншу сторону, світлодіод згасне.
Варто запам'ятати, що біт ACO регістра ACSR приймає значення:

- 0 - якщо AIN0 <AIN1
- 1 - якщо AIN0> AIN1

Лекція 6

Послідовні інтерфейси мікроконтролерів AVR

Питання:
1.Послідовний інтерфейс SPI
2.Послідовний інтерфейс I2C

1.Послідовний інтерфейс SPI

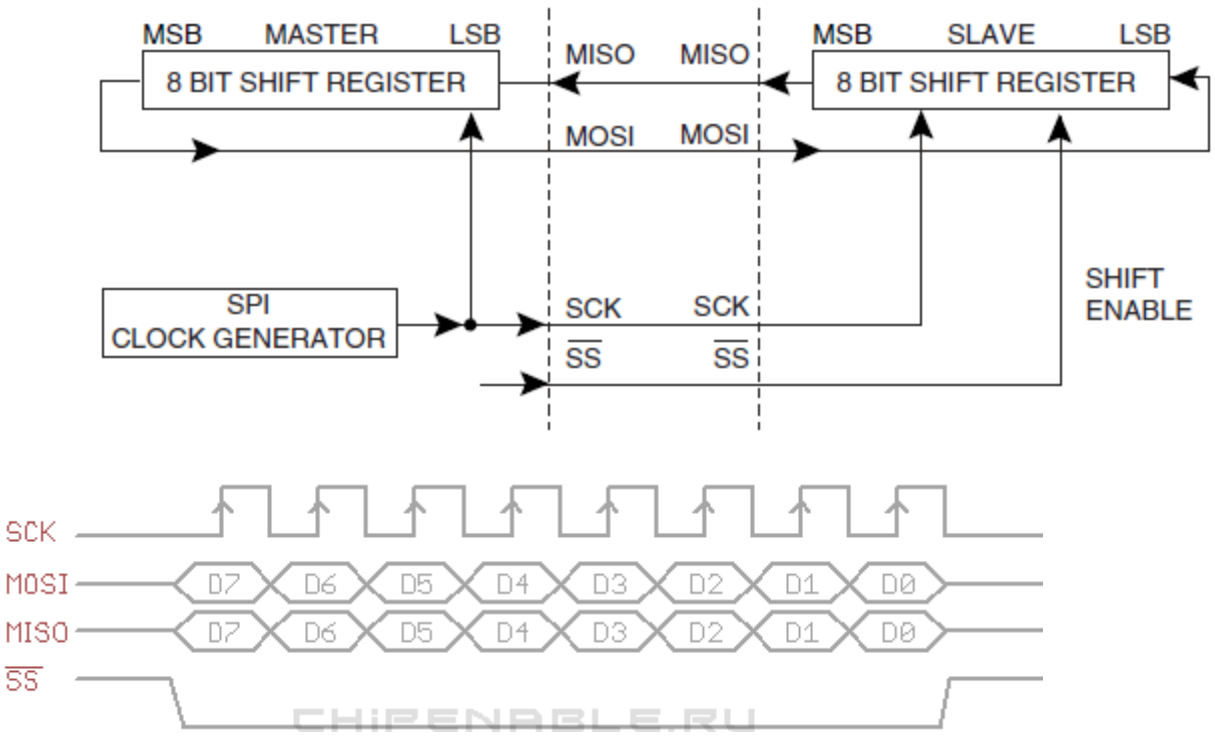
Мікроконтролери AVR мають у своєму складі модулі, що реалізують стандартні інтерфейси. Ці модулі використовуються для обміну даними з різними периферійними пристроями, наприклад, цифровими датчиками, мікросхемами пам'яті, ЦАП, АЦП, іншими микроконтроллерами і так далі. На прикладі мікроконтролера atmega16, ми розберемося, як працювати з модулем послідовного периферійного інтерфейсу або модулем SPI (serial peripheral interface).

SPI є чьотирьох синхронну шину, призначену для послідовного обміну даними між мікросхемами. Інтерфейс був розроблений компанією Motorola, але в даний момент використовується всіма виробниками. Даний інтерфейс відрізняють простота використання і реалізації, висока швидкість обміну і мала дальність дії.

При будь-якому обміні даними по SPI один з пристроїв є провідним (master'ом), а інше веденим (Slave'ом). Зазвичай (але не завжди) в ролі ведучого виступає мікроконтролер. Ведучий переводить периферійний пристрій в активний стан і формує тактовий сигнал і дані. У відповідь ведене пристрій передає ведучому свої дані. Передача даних в обидві сторони відбувається синхронно з тактовим сигналом.

Фізично SPI реалізується на основі зсувного регістру, який виконує і функцію передавача, і функцію приймача.

Принцип обміну даними по SPI проілюстрований на наступних картинках.

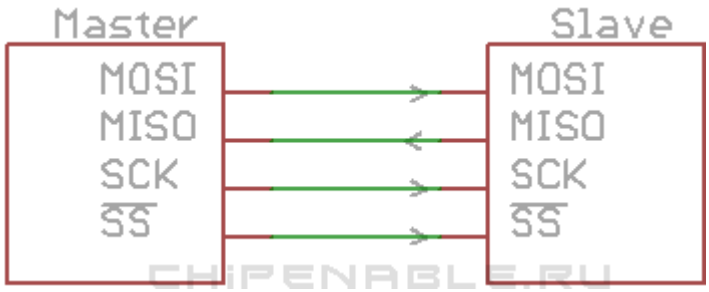


Сигнали, які використовуються даним інтерфейсом, мають таке призначення:

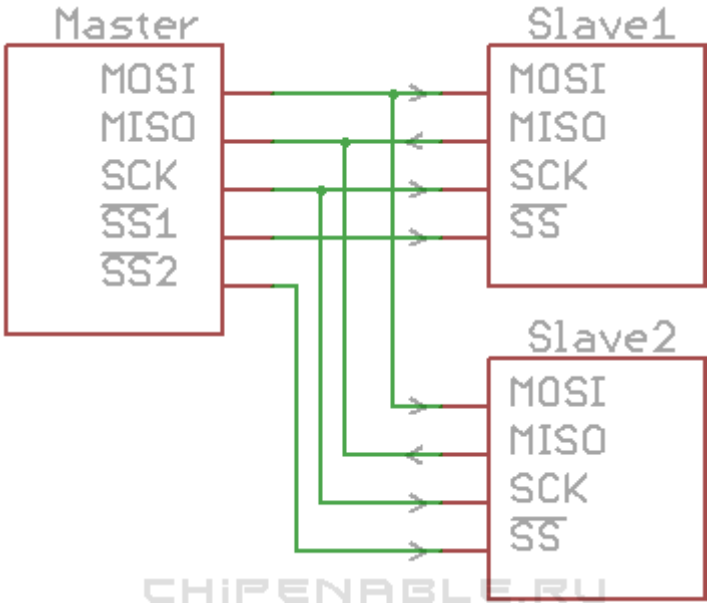
MOSI - Master Output / Slave Input. Вихід ведучого / вхід веденого. Служить для передачі

даних від провідного пристрою до веденого. **MISO** - Master Input / Slave Output. Вхід ведучого / вихід веденого. Служить для передачі даних від веденого пристрою до ведучого. **SLK** - Serial Clock.Сигнал синхронізації. Служить для передачі тактового сигналу всім відомим пристроїв. **SS** - Slave Select. Вибір веденого. Служить для вибору веденого пристрою. Виробники мікросхем часто використовують інші назви для цих сигналів. Альтернативні варіанти можуть бути такими: **MOSI** - DO, SDO, DOUT.**MISO** - DI, SDI, DIN. **SCK** - CLK, SCLK. **SS** - CS, SYNC.

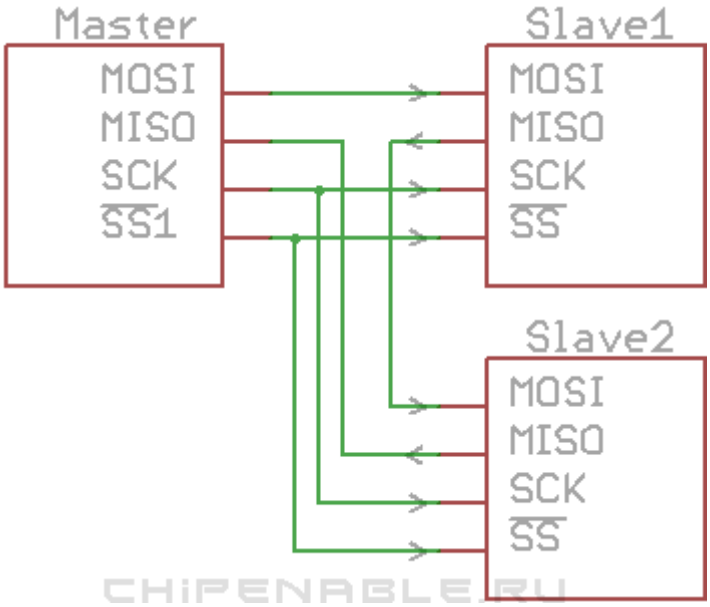
Схеми з'єднань по SPI
Типова схема з'єднання двох пристроїв по SPI виглядає так.



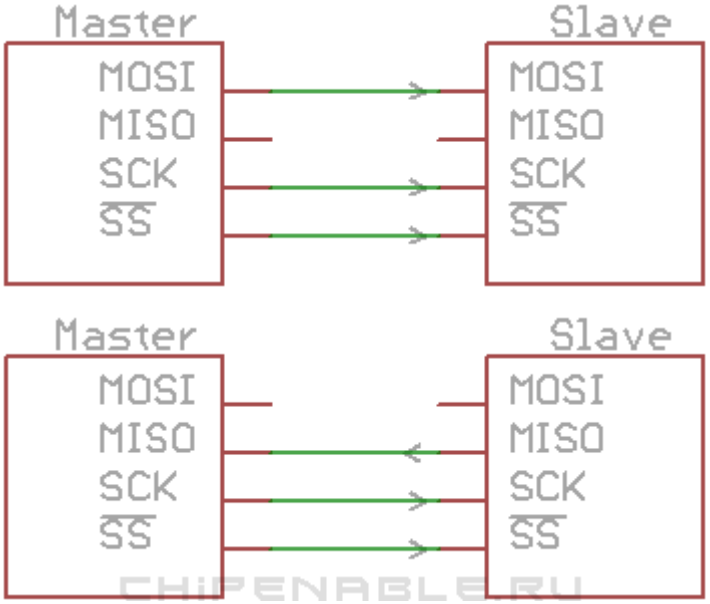
Також можливе підключення до ведучого пристрою кілька ведених пристроїв. Однак в будь-який момент часу обмін може відбуватися тільки з одним з них, інші повинні знаходитися в неактивному стані.



Виняток становить каскадна схема сполуки по SPI. При такому підключенні зсувні регістри пристроїв утворюють один великий регістр, і кількість ліній SPI залишається рівним 4-му. Правда, таке підключення підтримують далеко не всі мікросхеми.



Також можливий скорочений варіант схеми підключення, коли лінія MOSI або MISO не використовується. Тобто передача даних здійснюється тільки в одну сторону. Такі схема, наприклад, використовуються при підключенні до мікроконтролеру зовнішніх мікросхем ЦАП і АЦП.

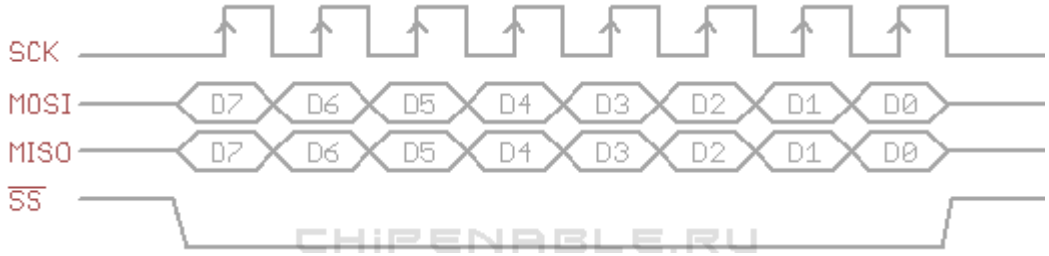


Протокол обміну по SPI

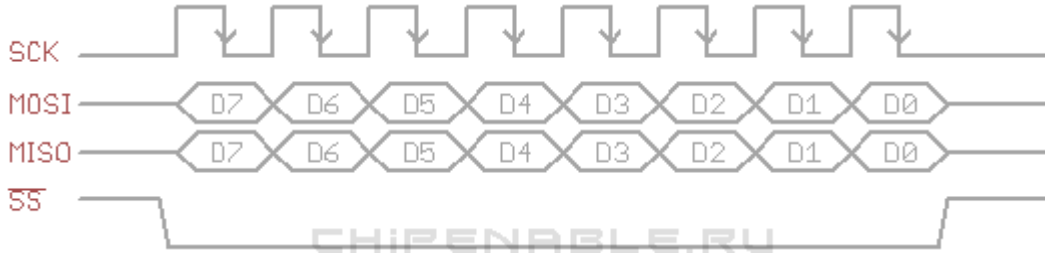
Протокол обміну по SPI аналогічний логіці роботи зсувного регістру і полягає в послідовному побітному виведення / введення даних за певними напрямами тактового сигналу. Установка даних і вибірка здійснюється по протилежних фронтах тактового сигналу.

Специфікація SPI передбачає 4 режими передачі даних, які відрізняються між собою співвідношенням фази і полярності тактового сигналу і переданих даних.

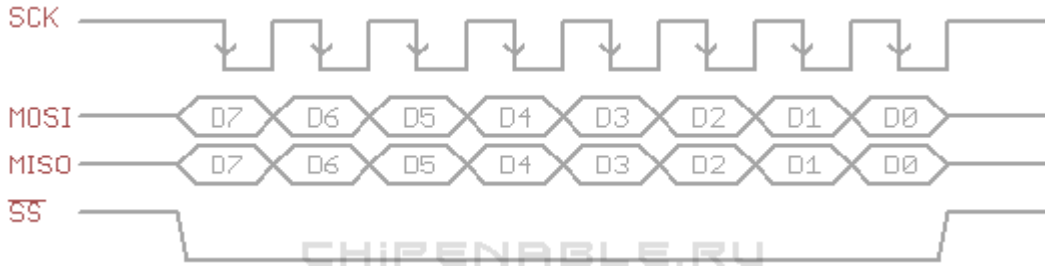
Ці режими описуються двома параметрами: **CPOL** - clock polarity. Полярність тактового сигналу - визначає вихідний рівень сигналу синхронізації **CPHA** - clock phase. Фаза тактового сигналу - визначає послідовність установки і вибірки даних. Малюнки нижче ілюструють всі чотири режими обміну SPI.**SPI mode 0: CPOL = 0, CPHA = 0.** Тактовий сигнал починається з рівня логічного нуля. Защелкивание даних виконується по наростаючому фронті.Зміна даних відбувається по падаючому фронту. Моменти защелкивание даних показані на малюнках стрілочками



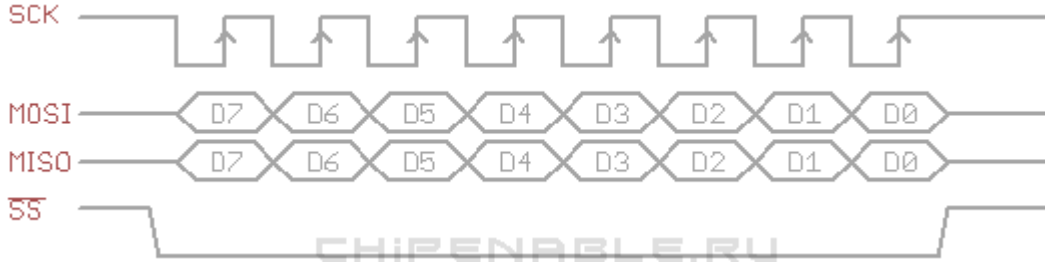
SPI mode 1: CPOL = 0, CPHA = 1. Тактовий сигнал починається з рівня логічного нуля. Зміна даних відбувається по наростаючому фронті.Защелкивание даних виконується по падаючому фронту.



SPI mode 2: CPOL = 1, CPHA = 0. Тактовий сигнал починається з рівня логічної одиниці. Защелкивание даних виконується по падаючому фронту.Зміна даних виконується по наростаючому фронті тактового сигналу.



SPI mode 3: CPOL = 1, CPHA = 1. Тактовий сигнал починається з рівня логічної одиниці. Зміна даних виконується по падаючому фронту тактового сигналу. Защелкивание даних виконується по наростаючому фронті.



Сучасні мікроконтролери підтримують усі чотири режими роботи SPI. Варто відзначити, що передача даних по SPI може відбуватися не тільки старшим бітом

вперед, але і молодшим. А кількість байт переданих за час утримання сигналу вибору (SS) нічим не обмежена і визначається специфікацією використовуваного веденого пристрою. Також в специфікації на ведене пристрій вказуються підтримувані режими роботи SPI, максимальна частота тактового сигналу, вміст переданих або отриманих даних.

Тепер ви маєте загальне уявлення про послідовне периферійному інтерфейсі і можна перейти до розгляду SPI модуля.

SPI модуль мікроконтролера AVR atmega16 використовує для своєї роботи 4 виведення - MOSI, MISO, SCK і SS. Коли модуль не задіяні, ці висновки є лініями портів введення / виводу загального призначення. Коли модуль включений, режим роботи цих висновків перевизначаються відповідно до наступної таблиці.

Pin	Direction, Master SPI	Direction, Slave SPI
MOSI	User Defined	Input
MISO	Input	User Defined
SCK	User Defined	Input
$\overline{SS}$	User Defined	Input

Якщо до мікроконтролеру підключено більше одного периферійного пристрою, в якості додаткових висновків вибору (SS), можна використовувати будь-які висновки загального призначення. При цьому штатний висновок SS повинен бути завжди правильно налаштований, навіть якщо він не використовується.

Регістри SPI модуля

У мікроконтролері atmega16 для роботи з модулем SPI використовуються три регістру: - керуючий регістр SPCR, - статусний регістр SPSR, - регістр даних SPDR. Всі три регістра восьмирозрядні. **Кофігурація модуля SPI встановлюється за допомогою регістра SPCR (SPI Control Register).**

Bit	7	6	5	4	3	2	1	0	
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

SPIE - дозволяє / забороняє переривання від модуля SPI. Якщо біт встановлено у 1, переривання від SPI дозволені. **SPE** - включає / вимикає модуль SPI. Якщо біт встановлено у 1, модуль SPI включений. **DORD** - визначає порядок передачі даних. Коли біт встановлений в 1, вміст регістра даних передається молодшим бітом вперед. Коли біт скинуто, то старшим бітом вперед. **MSTR** - визначає режим роботи мікроконтролера. Якщо біт встановлено у 1, мікроконтролер працює в режимі Master (ведучий). Якщо біт скинутий - в режимі Slave (ведений). Зазвичай мікроконтролер працює в режимі master. **CPOL** і **CPHA** - визначають в якому режимі працює SPI модуль. Необхідний режим роботи залежить від використовуваного периферійного пристрою.

Mode	CPOL	CPHA
SPI Mode 0	0	0
SPI Mode 1	0	1
SPI Mode 2	1	0
SPI Mode 3	1	1

SPR1 і **SPR0** - визначають частоту тактового сигналу SPI модуля, тобто швидкість обміну. Максимально можлива швидкість обміну завжди вказується в специфікації периферійного пристрою. **Статусний регістр SPSR (SPI Status Register) призначений для контролю стану SPI модуля**, крім того він містить додатковий біт керування швидкістю обміну.

Bit	7	6	5	4	3	2	1	0	
	SPIF	WCOL	–	–	–	–	–	SPI2X	SPSR
Read/Write	R	R	R	R	R	R	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

SPIF - прапор переривання від SPI. Він встановлюється в 1 після закінчення передачі байта даних. Якщо дозволені переривання модуля, одночасно з установкою цього прапора генерується переривання від SPI. Також цей прапор встановлюється в 1 при перекладі мікроконтроллера з режиму master в режим slave за допомогою виведення SS.

Скидання прапора відбувається апаратно, при виклику підпрограми обробки переривання або після читання регістра SPSR з подальшим зверненням до регістру даних SPDR. **WCOL** - прапор конфлікту записи. Прапор встановлюється в 1, якщо під час передавання даних відбувається спроба запису в регістр даних SPDR. Прапор скидається апаратно після читання регістра SPSR з подальшим зверненням до регістру даних SPDR. **SPI2X** - біт подвоєння швидкості обміну.Установка цього розряду в 1 подвоює частоту тактового сигналу SCK. Мікроконтролер при цьому повинен працювати в режимі master. Взаємозв'язок між бітами SPR0, SPR1, SPI2X і частотою тактового сигналу SCK показана в таблиці.

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

Де Fosc - тактова частота мікроконтролера AVR.

Для передачі і прийому даних призначений регістр SPDR (SPI Data Register) . Запис даних в цей регістр ініціює передачу даних SPI модулем. При читанні цього регістра, зчитується вміст буфера зсувного регістру SPI модуля.

Програмний код

Мінімальний програмний код для роботи з SPI модулем складається з двох функцій: - функції ініціалізації. - Функції передачі / прийому байта даних **Ініціалізація SPI модуля** Ініціалізація включає в себе конфігурацію висновків SPI модуля і керуючого регістра SPCR.

```
#define SPI_PORTX PORTB
#define SPI_DDRX DDRB

#define SPI_MISO 6
#define SPI_MOSI 5
#define SPI_SCK 7
#define SPI_SS 4

/ * Ініціалізація SPI модуля в режимі master * /
void SPI_Init ( void )
{
  / * Настройка портів введення-виведення
  всі висновки, крім MISO виходи * /
  SPI_DDRX | = (1 << SPI_MOSI) | (1 << SPI_SCK) | (1 << SPI_SS) | (0 << SPI_MISO);
  SPI_PORTX | = (1 << SPI_MOSI) | (1 << SPI_SCK) | (1 << SPI_SS) | (1 << SPI_MISO);

  / * Дозвіл spi, старший біт вперед, майстер, режим 0 * /
  SPCR = (1 << SPE) | (0 << DORD) | (1 << MSTR) | (0 << CPOL) | (0 << CPHA) | (1 << SPR
  SPSR = (0 << SPI2X);
}
```

6

7

8

9

0

Передача / прийом даних

Процес передачі / прийому даних за допомогою SPI модуля, що працює в режимі Master, складається з наступної послідовності дій: 1. установка низького логічного рівня на лінії SS 2. завантаження даних в регістр SPDR 3. очікування закінчення передачі (перевірка прапора SPIF) 4. збереження прийнятих даних (читання SPDR), якщо потрібно 5.повернення на 2-ий крок, якщо передані не всі дані 6. установка високого логічного рівня на лінії SS Нижче наведено кілька варіантів функції передачі / прийому даних. **Передача одного байта даних по SPI**

?

```
void SPI_WriteByte (uint8_t data)
{
  SPI_PORTX & = ~ (1 << SPI_SS);
  SPDR = data;
  while (! (SPSR & (1 << SPIF)));
  SPI_PORTX | = (1 << SPI_SS);
}
```

Передача і прийом одного байта даних по SPI

?

```
uint8_t SPI_ReadByte (uint8_t data)
{
  uint8_t report;
  SPI_PORTX & = ~ (1 << SPI_SS);
  SPDR = data;
  while (! (SPSR & (1 << SPIF)));
  report = SPDR;
  SPI_PORTX | = (1 << SPI_SS);
  return report;
}
```

0

Передача декількох байтів даних по SPI

* data - покажчик на масив даних, що передаються, а num - розмірність масиву

?

```
void SPI_WriteArray (uint8_t num, uint8_t * data)
{
  SPI_PORTX & = ~ (1 << SPI_SS);
  while (num -) {
    SPDR = * data ++;
    while (! (SPSR & (1 << SPIF)));
  }
  SPI_PORTX | = (1 << SPI_SS);
}

...

// Приклад використання:
uint8_t buf [3] = {12, 43, 98};

...
SPI_WriteArray (3, buf);
```

Передачі і прийом декількох байтів даних по SPI

* data - покажчик на масив даних, що передаються, а num - розмірність масиву. Прийняті дані будуть зберігатися в тому ж масиві.

?

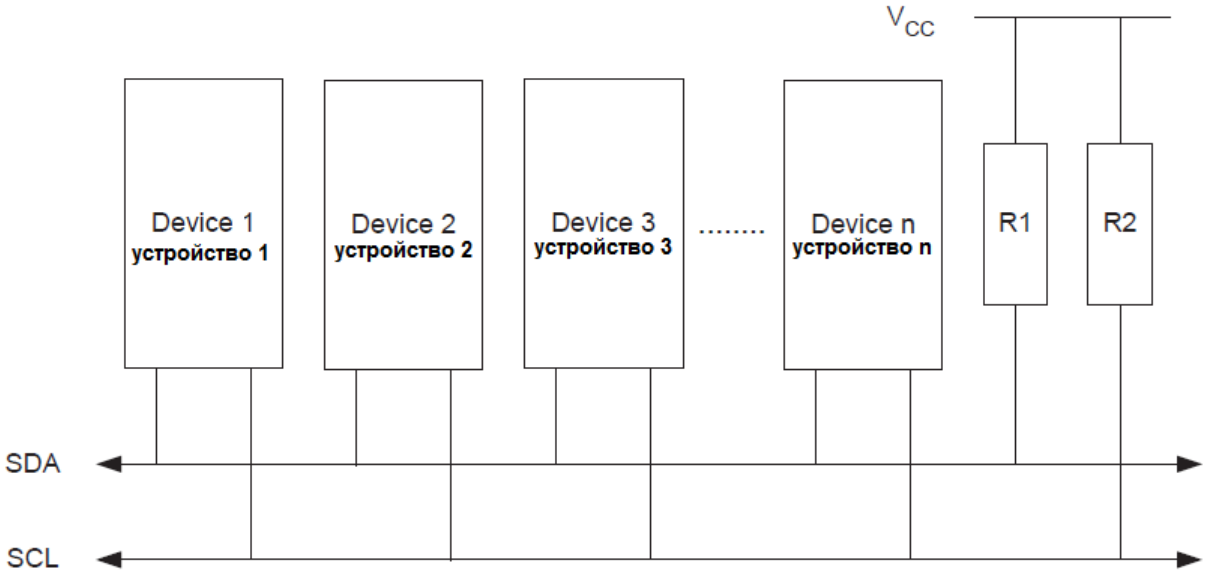
```
void SPI_ReadArray (uint8_t num, uint8_t * data)
{
  SPI_PORTX & = ~ (1 << SPI_SS);
  while (num -) {
    SPDR = * data;
    while (! (SPSR & (1 << SPIF)));
    * Data ++ = SPDR;
  }
  SPI_PORTX | = (1 << SPI_SS);
}
```

I2C інтерфейс являє собою дві двонаправлені лінії зв'язку - SDA і SCL. За SDA передаються дані, по SCL тактовий сигнал. Обидві лінії підтягнуті через резистори до плюса харчування.Фірма Philips за використання назви цього інтерфейсу вимагає ліцензійних відрахувань, тому в мікроконтролерах Atmel використовується власна назва TWI - two-wire interface («двухпроводной інтерфейс»).

Ось деякі переваги шини I2C:

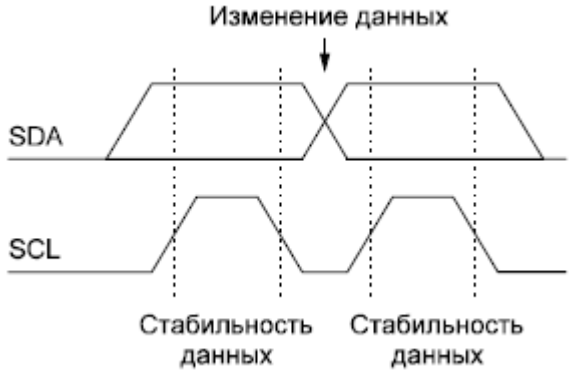
· Потрібно лише дві лінії - лінія даних (SDA) і лінія синхронізації (SCL). Кожен пристрій, підключений до шини, може бути програмно адресовано по унікальному адресою. У кожен момент часу існує просте ставлення провідний / ведений: провідні можуть працювати як провідний-передавач і ведучий-приймач. Шина дозволяє мати кілька провідних, надаючи кошти для визначення колізій і арбітраж, щоб запобігти пошкодженню даних в ситуації, коли два або більше провідних одночасно починають передачу даних. У стандартному режимі забезпечується передача послідовних 8-бітних даних зі швидкістю до 100 кбіт / с, і до 400 кбіт / с в «швидкому» режимі.

· Вбудований в мікросхеми фільтр придушує сплески, забезпечуючи цілісність даних.

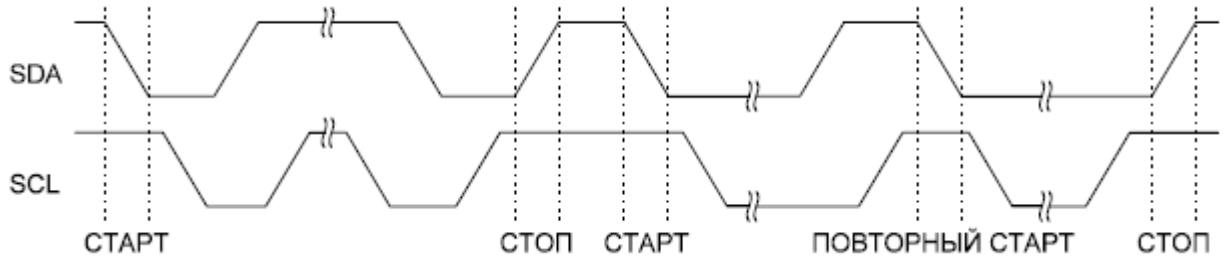


Формат передачі даних

Інтерфейс I2C є синхронним, тому кожен рухаючись біт даних на лінії SDA супроводжується імпульсом на лінії синхронізації SCL. Рівень даних повинен бути стабільним, коли на лінії синхронізації присутній лог. «1» Винятком для цього правила є генерація умов (СТАРТ / СТОП).

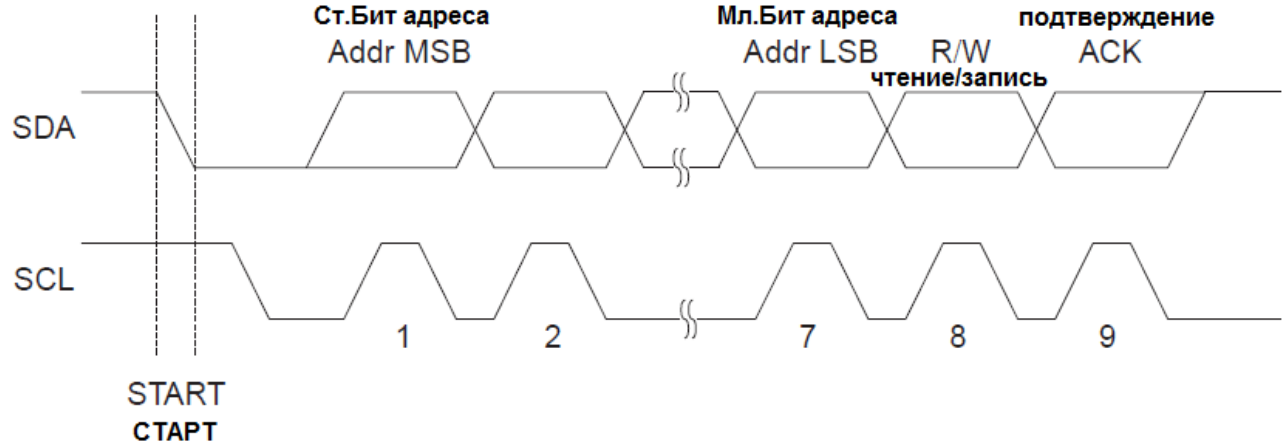


Провідний пристрій ініціює і закінчує передачу даних. Передача ініціюється, коли ведучий формує умова СТАРТ на шині, і припиняється, коли ведучий формує на шині умова СТОП.Між умовами СТАРТ і СТОП шина вважається зайнятою і в цьому випадку ніякої іншої майстер не може здійснювати управлінський вплив на шині. Існують особливі випадки, коли нову умову СТАРТ виникає між умовами СТАРТ і СТОП. Даний випадок іменується як "Повітряний старт" і використовується при необхідності ініціювати майстром новий сеанс зв'язку, не втрачаючи при цьому управління шиною. Після "Повторного старту" шина вважається зайнятою до наступного СТОП. Це ідентично поведінки після СТАРТ.



Формат адресного пакета

Всі передані адресні пакети по шині I2C складаються з 9 біт, в т.ч. 7 біт адреси, один біт управління для завдання типу операції читання або запису (R / W) і один біт підтвердження (ACK). Якщо біт R / W = 1, то буде виконана операція читання, інакше - запис. Якщо підлеглий розпізнає, що до нього відбувається адресація, то він повинен сформувати низький рівень на лінії SDA на 9-му циклі SCL (формування біта підтвердження). Якщо адресується підлеглий пристрій зайнято або з яких-небудь інших причин не може обслужити ведучий пристрій, то на лінії SDA необхідно залишити високий рівень під час циклу підтвердження. Ведучий після цього може передати умова СТОП або "Повторного старту" для ініціації нової пере-дачі.

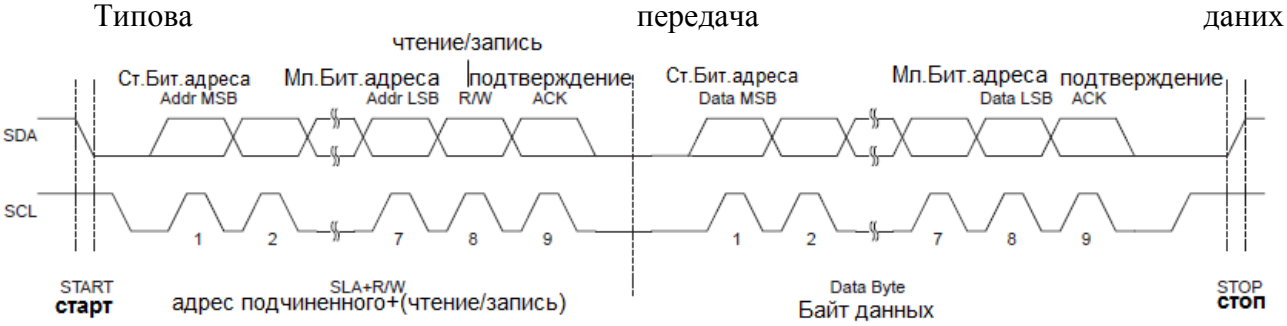


R / W - визначає подальший напрям передачі даних. Біт квітірованія - це відповідь веденого пристрою на прийнятий адресу. Якщо адреса розпізнано, ведений видає на лінію SDA низький рівень. В іншому випадку на лінії утримується високий рівень. ACR - біт підтвердження. Низький рівень означає відповідь від пристрою до якого звертається ведучий. Високий рівень говорить про те, що пристрій зайнятий.

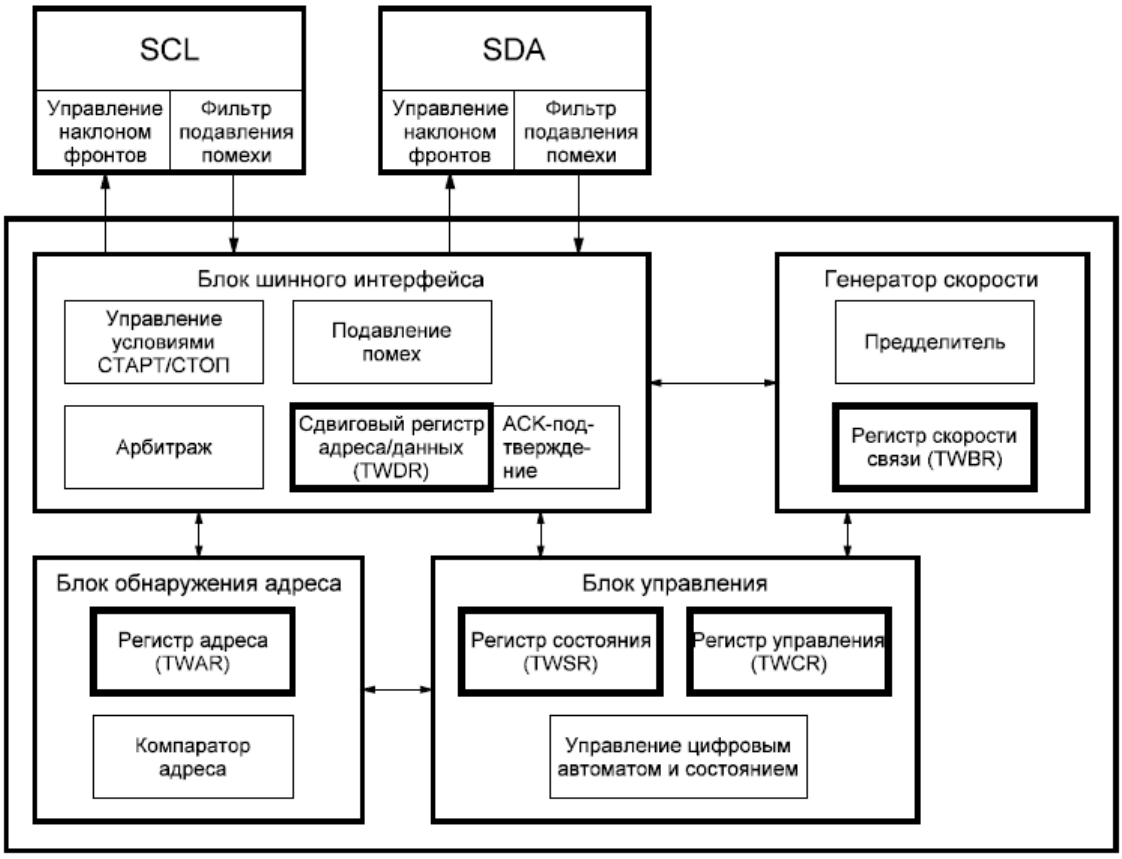
Пакети даних

пакети даних складаються з байта даних і біта квітірованія, тобто теж мають довжину 9 біт.Після прийому кожного байта даних, приймаючий пристрій (приймач) відповідає передавальному пристрою (передавача), встановлюючи на лінії SDA низький рівень (це і є біт квітірованія). Якщо приймаючий пристрій отримало останній байт або більше не може продовжувати прийом даних, воно повинно "залишити" на лінії SDA високий рівень.

Типова передача даних



Функціональна схема блоку шини I2C



Регістри I2C (TWI)

У виділених жирною лінією прямокутниках знаходяться регістри настройки I2C модуля в avr микроконтроллере. Для того щоб далі розбиратися з I2C модулем, потрібно ознайомитися з його регістрами. Розбір регістрів буде вестися на прикладі мікроконтролера atmega32. В інших мікроконтролерах можливі невеликі відмінності.

Регістр швидкості передачі TWBR (TWI Bit Rate Register) **Біт 7: 0**

Bit	7	6	5	4	3	2	1	0	
	TWBR7	TWBR6	TWBR5	TWBR4	TWBR3	TWBR2	TWBR1	TWBR0	TWBR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

-

біти цього регістра визначають частоту роботи модуля I2C. Частота також залежить від тактової частоти роботи мікроконтролера, і значення в молодших бітах (TWPS0 , TWPS1) регістра TWSR .

Частота SCL сигналу, тактова частота мікроконтролера, і значення регістрів TWBR і TWSRпов'язані наступним співвідношенням:

$$\text{SCL frequency} = \frac{\text{CPU Clock frequency}}{16 + 2(\text{TWBR}) \cdot 4^{TWPS}}$$

Регістр даних TWDR (TWI Data Register)

Bit	7	6	5	4	3	2	1	0	
	TWD7	TWD6	TWD5	TWD4	TWD3	TWD2	TWD1	TWD0	TWDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	1	1	1	1	1	1	1	

Біт 7: 0 - біти цього регістра зберігає дані, які ми або хочемо передати відомому, або отримали від ведучого.

Регістр адреси TWAR (TWI Address Register)

Bit	7	6	5	4	3	2	1	0	
	TWA6	TWA5	TWA4	TWA3	TWA2	TWA1	TWA0	TWGCE	TWAR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	1	1	1	1	1	1	1	0	

Біт 7: 1 - біти зберігання значення адреси, за якою буде відгукуватися мікроконтролер, перебуваючи в режимі веденого. **Біт 0** - біт дозволу на відгук під час спільних викликів.

Статусний регістр TWSR (TWI Status Register)

Bit	7	6	5	4	3	2	1	0	
	TWS7	TWS6	TWS5	TWS4	TWS3	–	TWPS1	TWPS0	TWSR
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	1	1	1	1	1	0	0	0	

Біт 7: 3 - б іти містять статусний код. Біти доступні тільки для читання, статусний код встановлюється TWI модулем апаратно, після виконання різних операцій. Наприклад, формування стану СТАРТ, передачі пакета даних і так далі. За значенням статусного коду можна судити про результат операції. Виконалася вона успішно чи ні. **Біт 2** - біт зарезервований і читається як 0. **Біт 1: 0** - біти впливають на частоту SCL (залежність наочна у формулі для обчислення SCL) Регістр управління TWCR (TWI Control Register)

TWPS1	TWPS0	Prescaler Value
0	0	1
0	1	4
1	0	16
1	1	64

Bit	7	6	5	4	3	2	1	0	
	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	–	TWIE	TWCR
Read/Write	R/W	R/W	R/W	R/W	R	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Біт 7 - біт прапора переривання TWI модуля. Цей біт встановлюється апаратно, коли TWI модуль завершує поточну операцію (формування стану СТАРТ, передачі адресного пакета і так далі). При цьому якщо встановлено біт глобального дозволу переривань (біт I регістраSREG) і дозволені переривання TWI модуля, то викликається відповідний обробник.

Біт TWINT очищається програмно, записом одиниці. При виконанні обробника переривання цей біт не скидається апаратно, як в інших модулях. Скидання прапора TWINT запускає роботу TWI модуля, тому всі операції з регістром даних, статусу або адреси, повинні бути виконані до його скидання.

Поки біт TWINT встановлено, на лінії SCL утримується низький рівень.

Біт 6 - біт дозволу підтвердження. Якщо біт TWEA встановлений в 1, TWI модуль формує сигнал підтвердження (ACK), коли це потрібно. А потрібно це в трьох випадках: провідне або ведене пристрій одержав байт даних, ведене пристрій одержав загальний виклик, ведене пристрій одержав свою адресу.

Біт 5 - прапор стану СТАРТ. Коли цей біт встановлюється в 1, TWI модуль перевіряє чи не зайнята шина і формує стан СТАРТ. Якщо шина зайнята, він буде чекати появи на ній стану СТОП і після цього видасть стан СТАРТ. Біт TWSTA повинен бути очищений програмно, коли стан СТАРТ передано.

Біт 4 - прапор стану СТОП. Коли цей біт встановлюється в 1 в режимі ведучого, TWI модуль видає на шину стан СТОП і скидає цей біт. У режимі веденого установка цього біта може використовуватися для відновлення після помилки. При цьому стан СТОП не формується, але TWI модуль повертається до початкового йдуть не до вашої станом. **Біт 3** - прапор конфлікту записи. Цей прапор встановлюється апаратно, коли виконується запис в регістр даних (TWDR) при низькому значенні біта TWINT . Тобто коли TWI модуль вже

виконує якісь операції. Прапор TWWCскидається апаратно, коли запис в регістр даних виконується при встановленому прапорі переривання TWINT .

Біт 2 - біт дозволу роботи TWI модуля. Коли біт TWEN встановлюється в 1, TWI модуль включається і бере на себе управління висновками SCL і SDA. Коли бітTWEN скидається, TWI модуль вимикається.

Біт 1 - біт зарезервований і читається як 0.

Біт 0 - дозвіл переривання TWI модуля. Коли біт TWIE і біт I регістра SREG встановлені в 1 - переривання модуля TWI дозволені. Переривання будуть викликатися при установці бітаTWINT .

```
#include "i2cmaster.h"
#define US_address 0xE4
#define Start_US 0x50

int main (void)
{
	uint8_t distance = 0xFF;
	unsigned char busy = 1;
	// start execution of program
	i2c_init(); // initialize I2C serial communication
	i2c_start(US_address + I2C_WRITE);
	// address I2C device ultrasonic sensor with write access
	i2c_write(Start_US); //start ultrasonic measurement
	i2c_stop(); // release I2C bus

	while (1)
	{
		busy = i2c_start(US_address + I2C_READ);
		// address I2C device ultrasonic sensor with read access

		if (busy == 0)
		{
			distance = i2c_readNak(); // read one byte (first echo result)
			i2c_stop(); // release I2C bus

			i2c_start(US_address + I2C_WRITE);
			// address I2C device ultrasonic sensor with write access
			i2c_write(Start_US); //start new ultrasonic measurement
			i2c_stop(); // release I2C bus
		}
		else
		{
			i2c_stop(); // release I2C bus
		}

		// insert your own code here

	}
	return 0;
```

ЛАБОРАТОРНА РОБОТА №1

ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ РОЗРОБКИ AVR STUDIO

Ціль: навчитися використати для написання програм інтегроване середовище розробки AVR Studio IDE (Integrated Development Environment).

Завдання: скомпілювати й налагодити програму в середовищі програмування AVR Studio.

ЗАГАЛЬНІ ВІДОМОСТІ

AVR Studio – професійне інтегроване середовище розробки, призначене для написання й налагодження прикладних програм для AVR мікропроцесорів у середовищі Microsoft Windows.

Початок роботи

При написанні програмного забезпечення (ПО) для мікроконтролера (МК) необхідно виконати стандартну послідовність дій:

- ✓ створення проекту;
- ✓ написання програми;
- ✓ зборка проекту;
- ✓ налагодження:
  - ➢ виправлення синтаксичних помилок;
  - ➢ симуляція роботи програми на ПК;
  - ➢ виправлення логічних помилок;
- ✓ програмування мікроконтролера (МК);
- ✓ перевірка роботи програми на реальній електронній платі.

Написання програми процес ітеративний, тобто при необхідності деякий набір дій прийде повторювати кілька разів до одержання задовільного результату.

При написанні ПО розроблювач може зштовхнутися із двома типами помилок: апаратні (помилка або несправність у принциповій схемі пристрою) і програмні (помилки в тексті ПО).

Програмні помилки при розробці ПО підрозділяються в такий спосіб:

- синтаксичні помилки;
- логічні помилки.

Синтаксичні помилки – помилки в тексті програми, які знаходить компілятор. Виправляються швидко й безболісно в більшості випадків. Програма при цьому не може бути скомпільована й запущена.

Логічні помилки – помилки в логіку роботи програми. Це той тип помилок при

якому ПО компілюється й може бути запрограмоване в МК, але воно працює не вірно (не тому що замислювалося). Ці помилки перебувають і усуваються за допомогою різноманітних допоміжних, отладочных засобів (симулятор, висновок отладочной інформації на консоль, апаратний отладчик і ін.).

СТВОРЕННЯ НОВОГО ПРОЕКТУ

При запуску AVR Studio відкривається стартова сторінка (Start page) див. Рис. 1.

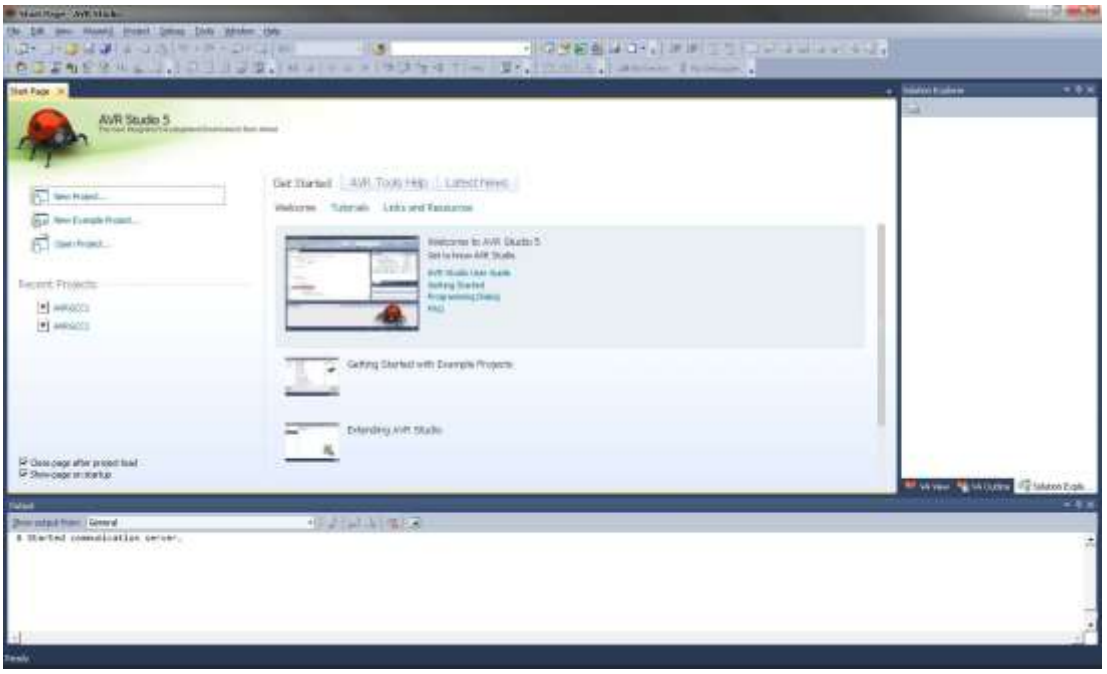


Рис. 1. Стартова сторінка AVR Studio

Дана сторінка дозволяє почати створювати новий проект (New Project),

відкрити проект із яким працювали раніше (Open Project), подивитися приклади проектів (New Example Projects), а також одержати різноманітну довідкову й допоміжну інформацію.

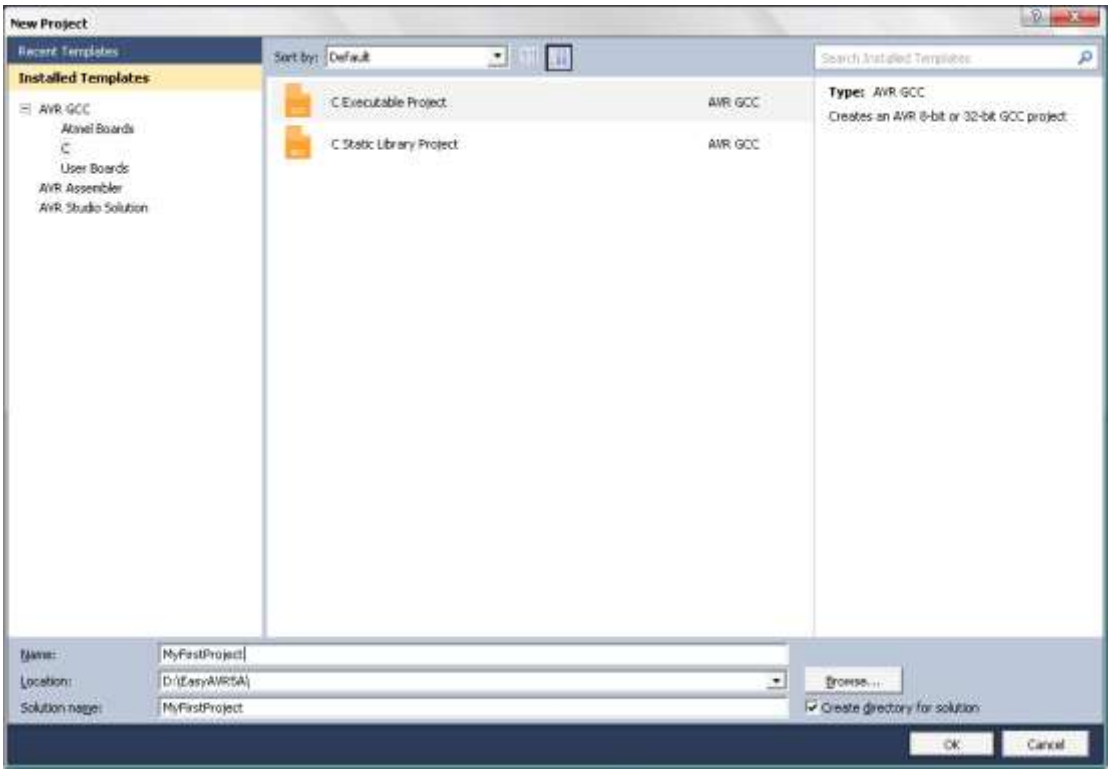


Рис. 2. Створення нового проекту

У вкладці встановлені шаблони (Installed Templates) вибираємо «З». А в допоміжному вікні праворуч - виконується программа, що, мовою З (З Executable Project).

У поле Name уводимо ім'я проекту. Ім'я проекту не повинне містити російських букв і пробілів. Поле Location містить шлях куди буде збережений проект (буде створена окрема папка при встановленому прапорі (Create directory for solution)).

Далі відкривається вікно вибору МК для якого пишеться ПО (див. Рис. 3).

У всіх лабораторних роботах ми будемо вивчати МК ATmega16 - вибираємо МК.

Після всіх перерахованих дій проект створений і можна приступати до написання програми.

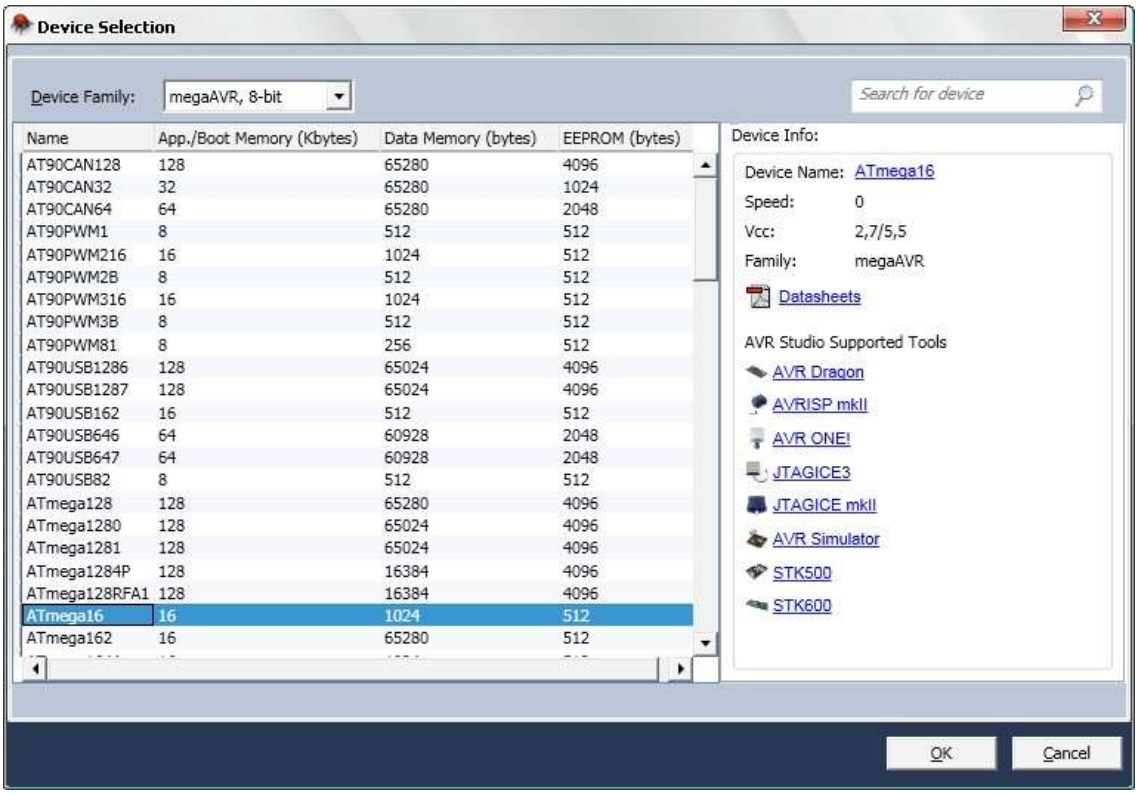


Рис. 3. Вікно вибору МК

ВВЕДЕННЯ ТЕКСТУ ПРОГРАМИ

Основне вікно розробки програми представлено на Рис. 4.

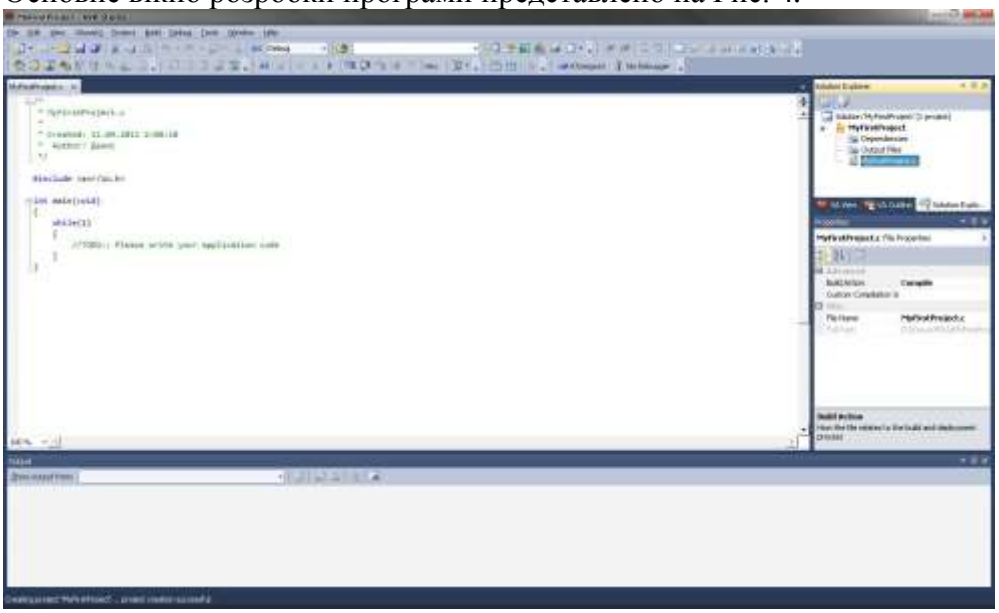


Рис. 4. Основне вікно розробки програми

У верхній частині вікна розташований - рядок меню й різні панелі інструментів (Toolbars).

У центрі вікна розміщене вікно для редагування/перегляду файлів проекту. У цей момент відкритий файл тексту програми з розширенням *.c

Праворуч розміщене вікно проекту (Solution Explorer). У даному вікні можна побачити файли підключені до проекту, які файли з'являються в результаті компіляції й ін.

Унизу розташоване вікно висновку діагностичної інформації (Output). У дане вікно середовище розробки виводить інформацію про процес компіляції й зборки проекту, а також інформацію про знайдені помилки.

При створенні проекту середовище розробки створює заготівлю вихідного тексту основної процедури ПО (функція **main**), а також довідковий блок коментарів.

При написанні програми зарезервовані слова виділяються синім кольором, коментарі - зеленим, основний текст - чорним і ін., тобто виробляється підсвічування синтаксису.

При написанні програми дуже рекомендується дотримуватися «гарного стилю написання коду»!:

- ✓ робити відповідні відступи в тексті для виділення логічних блоків коду (як горизонтальні - пробіли, так і вертикальні - порожні рядки);
- ✓ давати осмислені імена змінним і функціям на англійському мові;
- ✓ постачати програму коментарями.

Додаткові підказки:

- ✓ за відкриваючою дужкою негайно ставити закриваючу й убивати потрібний текст між ними;
- ✓ при написанні програми користуватися методом послідовного наближення, тобто писати програму вроздріб (завершеними блоками) і періодично перевіряти (компілюючи текст програми й/або перевіряючи його на реальному пристрої).

ЗБІРКА ПРОЕКТУ

Після написання тесту програми, його варто відкомпілювати й зробити зборку проекту. Для зборки проекту існує панель зборки (Build), див. Рис. 5.

Рис. 5. Панель зборки проекту

На панелі присутні 3 кнопки:

- ✓ зборка проекту (Build);
- ✓ зборка рішення (Build Solution);
- ✓ скасування (Cancel).

Всі панелі середовища розробки є що набудовуються, тобто їхній вид може відрізнятися від наведеного вище.

Зборка проекту (Build) складається із двох основних фаз:

✓ компіляція вихідних текстів програми написаних Вами (Compiling);

✓ підключення відкомпільованих раніше стандартних бібліотек функцій і т.д. - називається лінковка (Linking).

Компіляція – процес перекладу тексту програми, написаної мовою програмування, в об'єктний модуль, що містить машинні команди конкретного процесора.

Компоновщик (лінкер) – програма, що робить компонування: приймає на вхід один або кілька об'єктних модулів і збирає по них виконується модуль, що.

За результатами зборки проекту у вікні Output буде наданий звіт про хід роботи, а в Solution Explorer будуть показані файли отримані в результаті зборки. Програма втримується в ELF-файлі й HEX-файлі (вони дублюють один одного, а розроблювач використає необхідний з них).

НАЛАГОДЖЕННЯ

На написання тексту програми йде 20% часу, а інші 80% - займає процес налагодження проекту (Debug) і доведення його до стану випуску (Release).

Налагодження – етап комп'ютерного рішення завдання, при якому відбувається усунення явних помилок у програмі. Часто виробляється з використанням спеціальних програмних засобів - відлагоджувальників.

Виправлення синтаксичних помилок

Якщо при написанні ПО минулому допущені синтаксичні помилки, то компілятор не зможе відкомпілювати вихідний текст програми й видасть список знайдених помилок (Error List), див. Рис. 6.

Рис. 6. Помилки й попередження компілятора

У списку помилок (Errors) показується їхній опис і можливе місце розташування.

Також можуть бути присутнім попередження (Warnings) - місця в коді на які варто звернути увагу, тому що вони можуть викликати помилку під час виконання програми. І повідомлення (Messages).

Після вдалої зборки проекту можна приступитися до перевірки логіки роботи програми за допомогою симулятора або на реальному пристрої.

Симуляція роботи програми на ПК

Симуляція – моделювання процесу виконання програми мікроконтролером на персональному комп'ютері. Один із самих потужних методів налагодження програм.

Для керування процесом симуляції існує панель налагодження (Debug Toolbar), див Рис. 7.

Рис. 7. Панель налагодження проекту

Для керування ходом виконання програми призначені наступні кнопки:

	запуск/пауза налагодження програми (Start Debugging and Break)
	зупинка налагодження (Stop Debugging)
	дана стрілка показує команду, що буде виконана наступної (Show Next Statement)
	виконання команди із заходом у підпрограму
	(Step Into)
	виконання команди без заходу в підпрограму (Step Over)
	виконання коду до кінця поточної підпрограми (Step Out)
	виконання коду до місця розташування курсору (Run To Cursor)
	скидання - перехід програми в початковий стан (Reset)
	виконання програми без зупинок (Continue)
	пауза виконання програми (Break All)
	додаткові функції (малюнок може відрізнятися)

Для зупинки програми в певній крапці коду використаються, так звані, контрольні крапки.

Контрольна крапка – інструкція, у програмі дійшовши до якої виконання програми призупиниться. Установлена контрольна крапка відзначена червоним кружком.

Список установлених контрольних крапок (Breakpoints) можна одержати в додаткових функціях.

Для контролю значення змінних оголошених у програмі використається вікно Locals, див. Рис. 8. Контролюючи значення змінних ми можемо зрозуміти логіку роботи ЗД. (Кнопка перебуває в додаткових функціях).

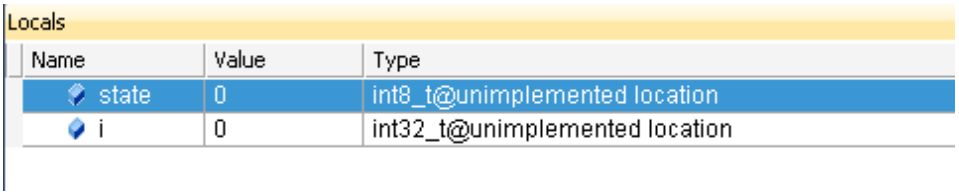


Рис. 8. Вікно контролю локальних змінних

У ході покрокового виконання програми (т.зв. трасування) розроблювач контролює стан МК, його периферії й модулів і т.д. у ході роботи програми. Для цього призначена панель налагодження AVR (AVR Debug Toolbar), див. Рис. 9.

Рис. 9. Панель налагодження AVR

На панелі є наступні команди:

	перегляд стану процесора (Processor View)
	перегляд регістрів загального призначення (Registers)
	перегляд пам'яті (Memory)
	перегляд периферії МК (I/O View)
	перегляд дізассемблованого коду (Disassembly)

У вікні стану процесора можна побачити наступне (див. Рис. 10)...

Програмний лічильник (Program Counter) – адреса машинної інструкції, що буде виконана в наступний такт (цикл) роботи МК.

Покажчик стека (Stack Pointer) – адреса вершини стека в пам'яті МК.

Регістр статусу (Status Register) – спеціальний регістр утримуючий статус результату виконання попередньої машинної інструкції. Інакше називається - регістр прапорів.

Лічильник циклів (Cycle Counter) – лічильник циклів минулих з моменту запуску програми.

Час зупинки (Stop Watch) – час у плині якого програма виконувалася.

Частота (Frequency) – тактова частота МК. Чим вище тактова частота, тим більше інструкцій МК може виконати за од. часу.

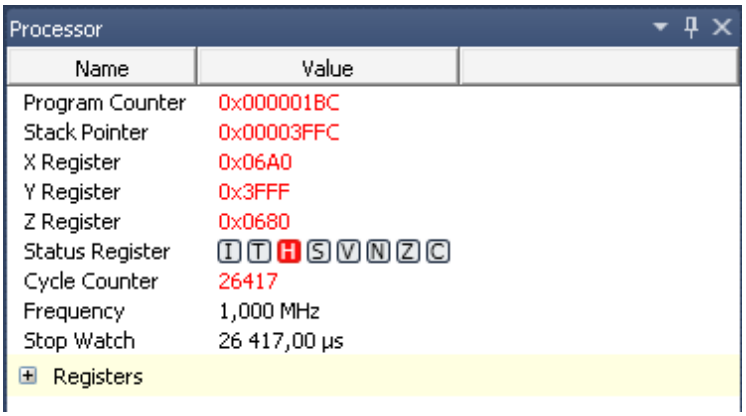


Рис. 10. Вікно стану процесора

Архітектура МК AVR припускає наявність 32-х 8-бітних регістрів загального призначення. Останні 6 регістрів формують, так звані, 16- бітні регістри непрямої адресації (X, Y, Z Registers) - вони використаються для доступу до пам'яті МК.

Вікно перегляду пам'яті дозволяє переглянути, а при необхідності й модифікувати, будь-яку пам'ять МК (пам'ять програм, пам'ять даних, EEPROM).

Вікно дизасемблювання показує, як програма мовою високого рівня була переведена в машинні коди компілятором.

Вікно перегляду периферії МК представлено на Рис. 11.

У даному вікні показані регістри, відповідальні за роботу убудованої в МК периферії, тобто порти введення/виводу, таймери, компаратори, комунікаційні інтерфейси й ін..

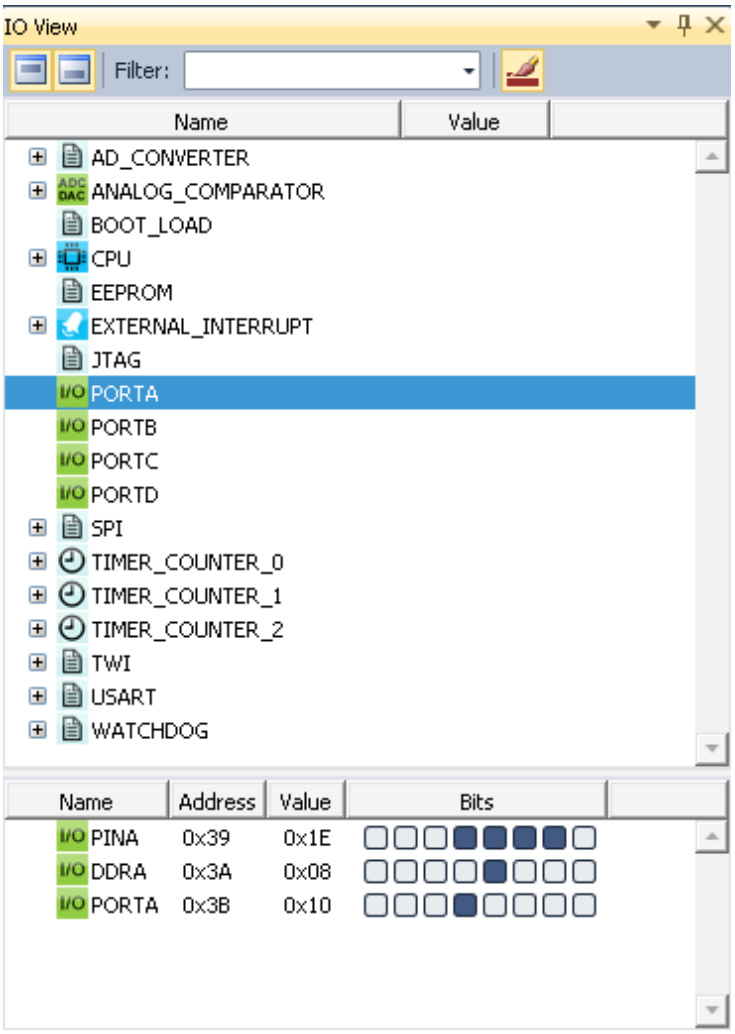


Рис. 11. Вікно перегляду периферії МК

ПРОГРАМУВАННЯ МІКРОКОНТРОЛЕРА

Коли програма налагоджена й працює відповідно до задуманої ідеї її можна прошивати в МК і тестувати на реальному МК у реальному оточенні.

Для прошивання МК може бути використана програма AvrFlash, див. Рис. 12.

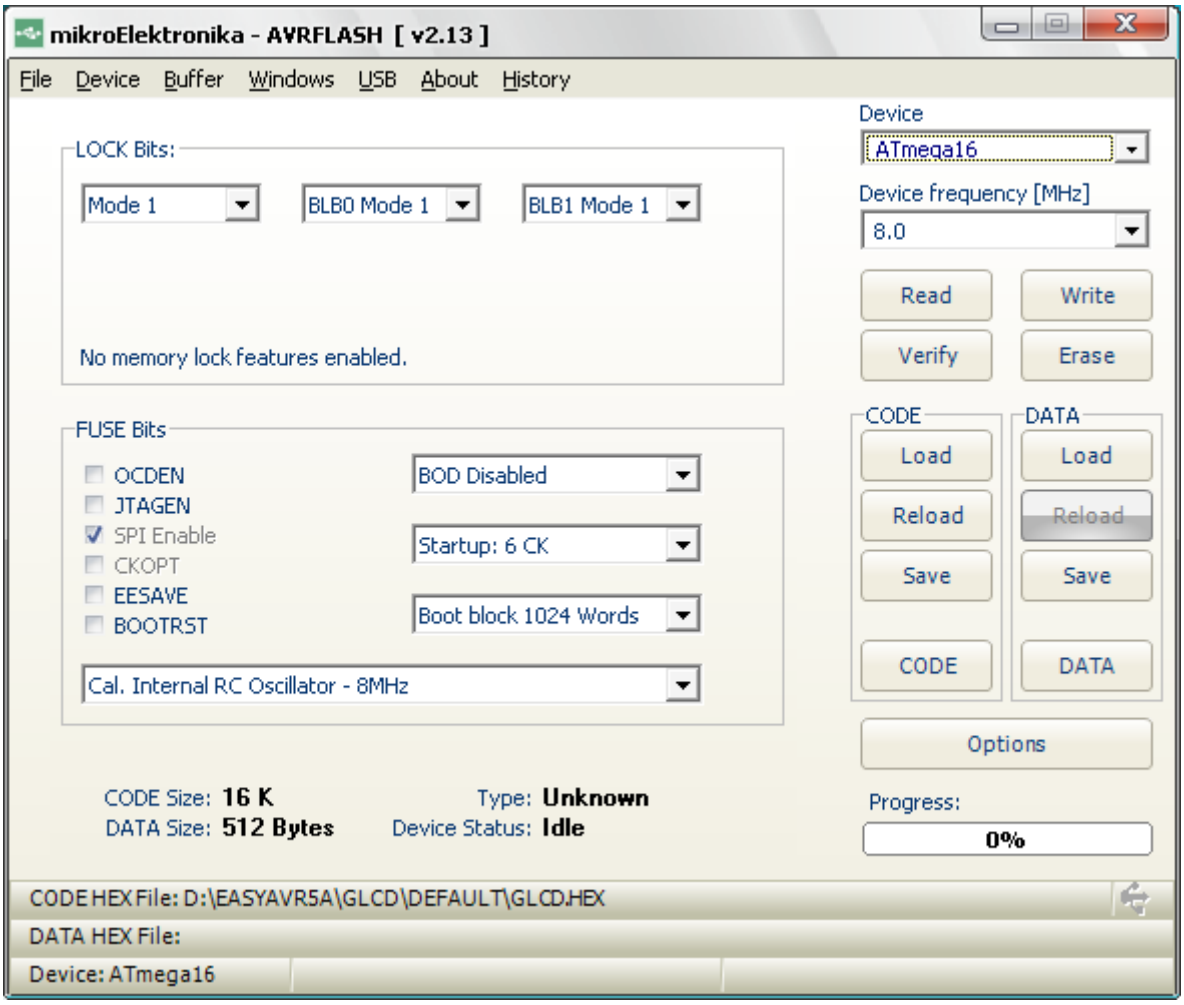


Рис. 12. Прошивання МК

ПОЛЯ В ОБЛАСТІ «LOCK BITS» НЕ ЗМІНЮВАТИ!!!

Це спеціальні біти призначені для уведення МК у різні режими (тільки читання, неможливість скидання й т.п.). Зміна даних полів приводить у деяких випадках до блокування МК (неможливо стерти й т.п.). Використати їх можна тільки з розумінням що відбувається.

Область Fuse Bits - треба виставити робочу частоту і її джерело (випадаючий список унизу групи). Device - виставити використовуваний МК. Device Frequency - частота на якій МК працює.
Область «CODE» - відповідає за роботу з кодом програми:

- ✓ LOAD - завантажити код програми на згадку ПК із HEX-файлу (*.hex);
- ✓ RELOAD - перезавантажити код програми на згадку ПК;
- ✓ SAVE - зберегти код програми на ПК.

Кнопки READ, WRITE, ERASE, VERIFY відповідають за читання, запис, стирання й перевірку МК відповідно.

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке AVR Studio.

2. Яка стандартна послідовність дій при розробці програми для мікроконтролера.
3. Що таке зборка.
4. Чим компілювання відрізняється від лінкування.
5. Що таке симуляція.
6. Навіщо роблять налагодження програми.
7. Що таке контрольна крапка.
8. Що таке дизасемблювання.
9. Навіщо призначена вкладка I/O View.

Текст програми

```
// Підключаємо зовнішні бібліотеки
#include <AVR/io.h>
#include <stdint.h>
#include <util/delay.h>


// Основна програма
int main(void)
{
unsigned int line;
unsigned int line;
unsigned int line;
unsigned int line;
short i;

// Налаштовуємо порти введення/виводу
DDRA   = 0xFF;
DDRB   = 0xFF;
DDRC   = 0xFF;
DDRD   = 0xFF;

// Вічний цикл
while (1)
{
// Формуємо вихідну картину
line       = 0b1100000000000000;
line       = 0b0110000000000000;
line       = 0b0110000000000000;
line       = 0b1100000000000000;

// Виводимо її на екран
for (i = 0; i < 20; i++)
{
// Вивід на світлодіоди
PORTA = line;
PORTB  = line;
PORTC        = line;
PORTD        = line;

// Зсування зображення line = line >> 1; line = line >> 1;
line = line >> 1; line = line >> 1;

// Затримка
_delay_ms(300);
}
}
}
```

Текст програми

```
// Константи для включення індикаторів
#define DIS3 0x08
#define DIS2 0x04
#define DIS1 0x02
#define DIS0 0x01

// Затримка зміни лінії
#define delay_const 400

// Підключаємо використовувані бібліотеки
#include <AVR/io.h>
#include <util/delay.h>

void move_line(void)
{
// Виводимо першу лінію
PORTC = 0x30;
_delay_ms(delay_const);

// Виводимо другу лінію
PORTC = 0x06;
_delay_ms(delay_const);

// Виключаємо висновок
PORTC = 0x00;
}

// Основна програма
int main(void)
{
// Настроювання портів
DDRA  = 0xFF;
DDRB  = 0xFF;
DDRC  = 0xFF;
DDRD  = 0xFF;

// Висновок лінії, що біжить
//  (вічний цикл) while(1)
{
// Включаємо перший символ
PORTB = DIS3;

// Зсуваємо лінію
move_line();

// Включаємо перший символ
PORTB = DIS2;
```

```
// Зсуваємо лінію
move_line();

// Включаємо перший символ
PORTB = DIS1;

// Зсуваємо лінію
move_line();

// Включаємо перший символ
PORTB = DIS0;

// Зсуваємо лінію
move_line();
}
}
```

Лабораторна робота № 2 ПРОГРАМА УПРАВЛІННЯ СВІТЛОДІОДАМИ

Мета роботи: ознайомитись з принципом управління дискретними пристроями, що приєднані безпосередньо до портів мікроконтролера, а також програмним забезпеченням WinAVR, Codevision AVR, Flowcode for AVR. Набути навиків створення та компіляції власної програми для мікроконтролера.

Обладнання: навчально-відлагоджувальна плата AVR-Easy; мікроконтролери ATtiny2313, ATmega8, ATmega16, Atmega8515; середовища програмування WinAVR, Codevision AVR, Flowcode for AVR; внутрішньосхемний програматор; програма для прошивки мікроконтролерів AVR8 Bum-O-Mat.

Теоретичний матеріал

Порти введення / виведення МК AVR мають число незалежних ліній "Вхід / Вихід" від 3 до 53. Кожен розряд порту може бути запрограмований на введення або на виведення інформації. Вихідні драйвери забезпечують струмову навантажувальну здатність 20 мА на лінію порту при максимальному значенні 40 мА, що дозволяє, наприклад, безпосередньо підключати до мікроконтролера світлодіоди і біполярні транзистори. Загальна струмова навантаження на всі лінії одного порту не повинна перевищувати 80 мА (всі значення наведено для напруги живлення 5 В).

Для роботи з будь-яким портом «x» існують три регістра управління:DDRx, PORTx, PINx.

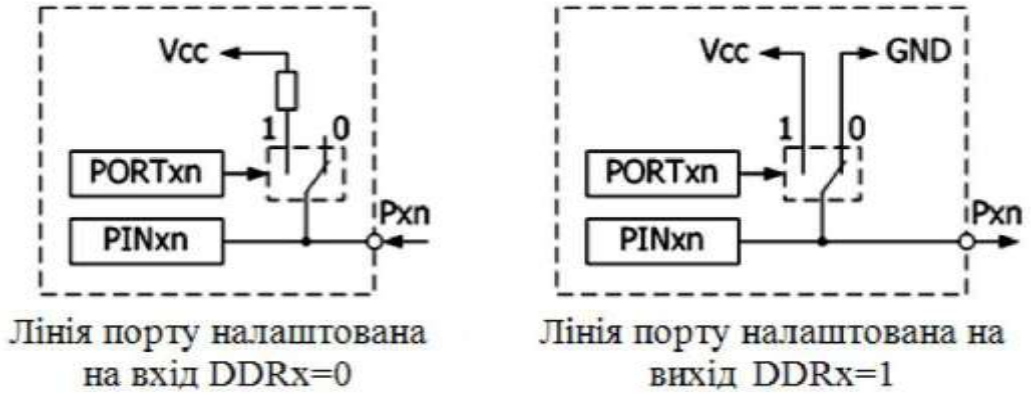


Рис. 2.1. Схеми налаштування портів на режими введення / виведення

DDRx (The Port X Data Direction Register) - визначає напрям передачі даних кожної лінії порту X: 1 - вихід, 0 - вхід. Регістр DDRx містить 8 біт DDx0 .. DDx7, кожен з яких відповідає за свою лінію порту, яка збігається з номером Px0 .. Px7.

PORTx (The Port X Data Register) - регістр даних порту «x». Принцип роботи з цим регістром залежить від того, в якому режимі, входу або виходу працює лінія порту. Якщо лінія порту працює на вхід, то біти даного регістра відповідають за підключення до лінії внутрішнього опору, який підтягує напругу лінії до напруги живлення. Якщо лінія порту працює на вихід, то біти даного регістра керують

станом вихідної лінії.

PINx (The Port X Input Pins Address) - діапазон адрес, читання по яким надає доступ до інформації з буферних регістрів на вході порту «x». Даний регістр призначений тільки для зчитування станів ліній порту.

Плата містить 32 світлодіода, підключених до портів A, B, C, D мікроконтролера. Для того щоб засвітити світлодіод необхідно ввімкнути потрібний порт перемикачем SW6 і записати '1 'в потрібний розряд. Тобто, низький високий рівень на портах мікроконтролера вмикає світлодіод, низький вимикає.

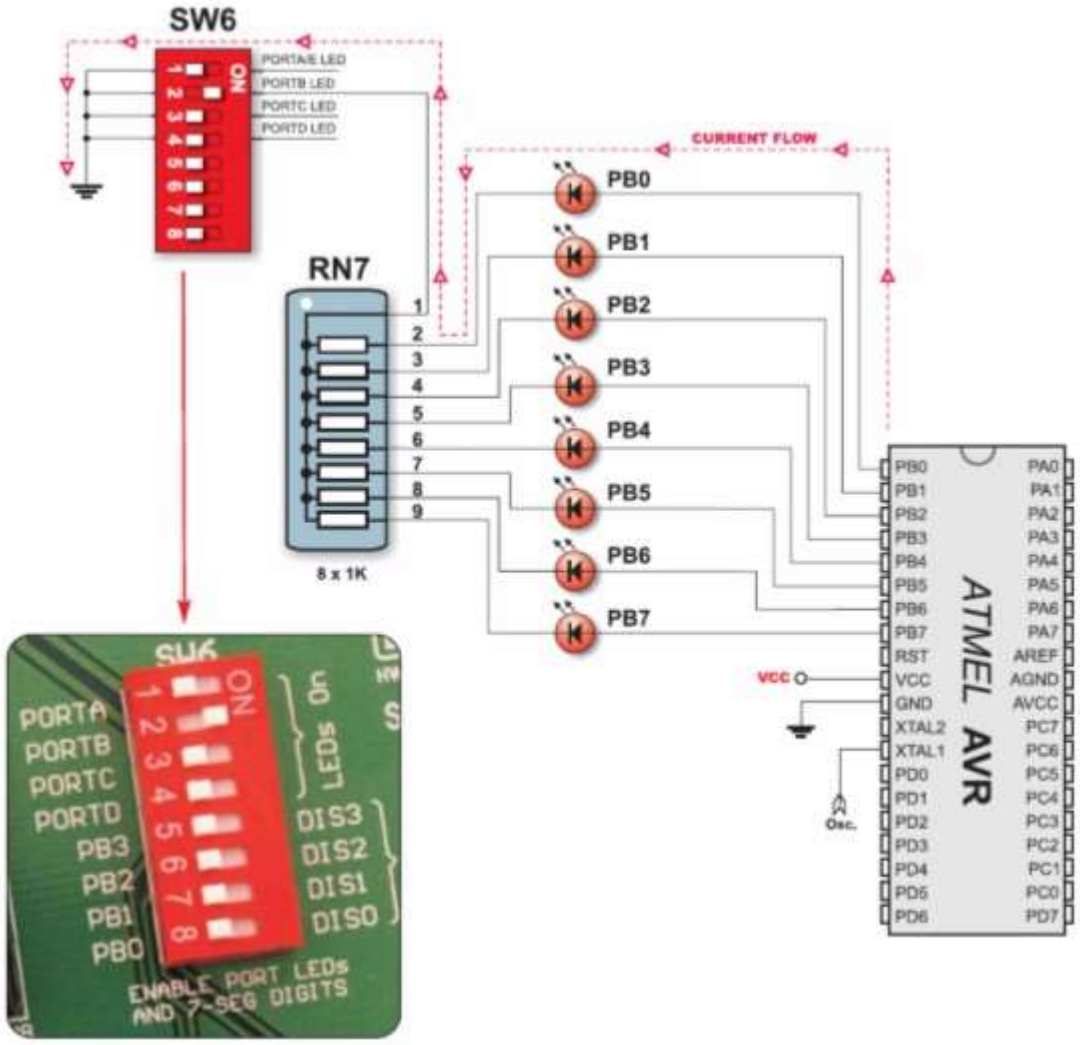


Рис. 1. Схема ввімкнення світло діодів

При виконанні лабораторних робіт для розробки програм рекомендовано використання AVR Studio.

Приклад програми миготіння двома світлодіодами порту B

```
# define F_CPU 8000000UL            // Вибираємо частоту МК
#include <avr/io.h>                 // Для роботи с портами
#include <util/delay.h>             // для затримки
```

```
int main(void)
{
    // 2 и 3 виводи порту B як вихід
    DDRB = 0b00001100;                    // двійковий код
    DDRB = 0x0C;                          // 16 - вий код
    DDRB = 12;                            // 10 - вий код
    DDRB |= ( 1 << 2 ) | ( 1 << 3 );      //     розряди 2 та 3 порта B на вихід

    // на 2 и 3 виводи подаємо логічну 1 ( + 5B )
    PORTB |= ( 1 << 2 ) | ( 1 << 3 );

    while(1)
    {
        // мигаємо 2 и 3 виводами
          PORTB ^= 0b00001100;
        _delay_ms(200);
    }
}
```

Для затримки використовуємо функцію _delay_ms (). Функція _delay_ms () формує затримку в залежності від переданого їй аргументу, вираженого в мілісекундах (в одній секунді 1000 мілісекунд). Максимальна затримка може досягати 262.14 мілісекунд. Якщо користувач передасть функції значення більш 262.14, то відбудеться автоматичне зменшення дозволу до 1/10 мілісекунди, що забезпечує затримки до 6.5535 секунд. Функція _delay_ms () міститься в файлі delay.h, тому нам буде необхідно підключити цей файл до програми. Крім того, для нормальної роботи цієї функції необхідно вказати частоту, на якій працює мікроконтролер, в герцах.

В наступній програмі організований нескінченний цикл за допомогою оператора безумовного переходу "goto".

```
  #define F_CPU 8000000UL   // вказуємо частоту в герцах

  #include <avr / io.h>
  #include <util / delay.h>

  int main ( void ) {        // початок основної програми

  DDRD = 0xff;                    // все висновки порту D конфігурувати як виходи

start:                            // мітка для команди goto start

  PORTD | = _BV (PD1);// встановити "1" (високий рівень) на виведення PD1,
                                  // Запалити світлодіод
```

```
  _delay_ms (250);          // чекаємо 0.25 сек.

  PORTD & = ~ _BV (PD1);// встановити "0" (низький рівень) на виведення
PD1,
                          // Погасити світлодіод

  _delay_ms (250);          // чекаємо 0.25 сек.

goto start;                 // перейти до мітки start

  }                         // Закриває дужка основної програми
```

У бібліотеці avr/io.h знаходяться деякі функції вводу-виводу та описи регістрів МК. Без цієї бібліотеки не обходиться жодна програма для AVR. У бібліотеці util/delay.h розміщені функції затримки (паузи).

Головна функція - main. Мікроконтролер виконує її після скидання або ввімкнення живлення. Вона закінчується нескінченним циклом та поверненням нуля.

Перш за все відбувається налаштування портів. Команда DDR* вказує напрям роботи. При передачі команді «1» - відповідний пін порту налаштовується на «вихід». При передачі «0» - відповідний пін порту налаштовується на «вхід». Команда PORTx налаштовує стан порту на початку програми.

Далі йде нескінченний цикл for(;;). На відміну від комп'ютерної програми, програма мікроконтролера працює у нескінченному циклі до вимкнення живлення або скидання МК.

Функція _delay_ms(10) з бібліотеки util/delay.h організує затримку 10 мілісекунд. Написати _delay_ms(500) не можна, бо максимальне значення аргументу 262.14 мс, поділене на тактову частоту в МГц. Безпечне значення для усіх мікроконтролерів AVR та тактових частот є 10 мс. Щоб зробити затримку 0, 5 с (500 мс), потрібно повторити 50 разів затримку 10 мс.

Для того, щоб не всі світлодіоди блимали, то необхідно вмикати лише потрібні світло діоди, наприклад, другий, третій та сьомий світлодіоди. Для цього треба виставляти в “1” лише біти 1, 2 та 6 (згадаємо, що перший світлодіод під'єднаний до біту 0), інші біти - в “0”. У програму потрібно вписати деяке число у шістнадцятковій системі числення (HEX). Приставка “0x” означає, що число записане у шістнадцятковому вигляді. Щоб його отримати, запишемо значення всіх восьми бітів, починаючи з найстаршого.

Таблиця 2.1

Приклади кодування станів поту в шістнадцятко вій системі

Біти								Число у HEX- форматі
7	6	5	4	3	2	1	0	
0	1	0	0	0	1	1	0	0x46
1	1	0	1	1	0	0	1	0xD9
0	1	0	1	1	1	1	1	0x5F

Розділимо біти на групи по 4 і замінимо кожну групу однією цифрою згідно таблиці. Отримаємо потрібне число.

Таблиця 2.2

Таблиця відповідності двійкових значень шістнадцятковим

Двійкове число				Шістнадняткова цифра	Двійкове число				Шістнадняткова цифра
0	0	0	0	0	1	0	0	0	8
0	0	0	1	1	1	0	0	1	9
0	0	1	0	2	1	0	1	0	A
0	0	1	1	3	1	0	1	1	B
0	1	0	0	4	1	1	0	0	C
0	1	0	1	5	1	1	0	1	D
0	1	1	0	6	1	1	1	0	E
0	1	1	1	7	1	1	1	1	F

Лінійка світлодіодів знаходиться у двох станах по черзі. У першому стані усі світлодіоди горять (0xFF), у другому усі погашені (0x00). Загалом, якщо в обох станах значення біту рівне “1”, відповідний світлодіод завжди горить, якщо “0”, то не горить, а якщо значення біту змінюється, то світлодіод блимає.

Хід роботи

1. Розібратися з принципом роботи тестової програми та призначенням кожного оператора у програмі.
2. Створити свою програму згідно з індивідуальним завданням та скомпілювати її.
3. Під'єднати внутрішньосхемний програматор до плати та комп'ютера. Увімкнути плату.
4. Запрограмувати МК.
5. Перевірити правильність виконання програми.

Індивідуальні завдання

Створити програму для свого типу мікроконтролера, яка виконує наступні функції: світлодіоди з вказаними номерами увімкнені, вимкнені або блимають згідно з таблицею.

Таблиця 2.3
Варіанти індивідуальних завдань

Варіант	Увімкнені	Вимкнені	Блимають
1	1,2	3,4	5, 6, 7, 8
2	3,6	2,7	1, 4, 5, 8
3	3, 6, 8	2,5	1, 4, 7
4	2,5	1, 7, 8	3, 4, 6
5	2,3	4,5	1, 6, 7, 8
6	2, 3, 4	1,5	6, 7, 8
7	1, 5, 8	3,6	2, 4, 7
8	2,4	6,8	1, 3, 5,7
9	1,7	3,5	2, 4, 6, 8
10	4,5	3,6	1, 2, 7, 8
11	1	2, 4, 6	3, 5, 7, 8
12	1,8	2,7	3, 4, 5,6
13	3, 4, 8	6,7	1, 2, 5
14	6, 7, 8	3	1, 2, 4, 5
15	1, 2, 8	3, 6, 7	4, 5, 6

Контрольні запитання

1. Кодування чисел в двійкову та шістнадцяткову системи числення;
2. Час виконання команд (поняття такту, машинного циклу);
3. Призначення портів введення/виведення AVR-мікроконтролерів;
4. Структура порту введення/виведення AVR-мікроконтролерів;
5. Регістри керування портами введення/виведення AVR-мікроконтролерів та їх призначення.

Зміст звіту

1. Тема та мета роботи.
2. Перелік використаного обладнання.
3. Стислий зміст теоретичних відомостей.
4. Лістинг власної програми з детальним поясненням кожного рядка.
5. Відповіді на контрольні запитання.
6. Висновки.

Лабораторна робота № 3 ОРГАНІЗАЦІЯ ДИНАМІЧНОЇ ІНДИКАЦІЇ

Мета роботи: ознайомитись з портами вводу-виводу мікроконтролера, принципом обробки сигналів дискретних датчиків. Набути навичок відображення інформації за допомогою світлодіодних індикаторів.

Обладнання: навчально-відлагоджувальна плата AVR-Easy; мікроконтролери ATtiny2313, ATmega8, ATmega16, Atmega8515; середовища програмування WinAVR, Codevision AVR, Flowcode for AVR; внутрішньосхемний програматор; програма для прошивки мікроконтролерів AVR8 Burn-O-Mat.

Теоретичний матеріал

Дуже часто МК використовується не тільки для керування роботою конструкції, але й для того, щоб повідомити що-небудь користувачеві. Наприклад, електронний годинник, крім власне відліків часу, повинен його ще відображати, а також дозволяти змінювати покази (встановлювати точний час). Якщо вся "інформація" зводиться до мигання парою світлодіодів, яких-небудь спеціальних зусиль з відображення інформації з боку розробника конструкції не вимагається, але якщо таких світлодіодів виявляється два-три десятки, тут вже потрібне застосування додаткових засобів (як апаратних, так і програмних). Як правило, в цьому випадку відображення інформації виконують у режимі динамічний індикації - це найбільш економний за кількістю використовуваних ліній спосіб.

Найбільш часто динамічну індикацію застосовують при здійсненні відображення інформації на семисегментних індикаторах, в яких стилізоване зображення цифр (і деякого набору букв) складають із семи лінійних сегментів, розташованих у вигляді цифри вісім (рис. 2.3). Висвічування сегмента, що вибирається, чи групи сегментів при отриманні зображення знаку забезпечується ввімкненням їх в коло проходження струму.

Рис 2.3. Зображення знаків на основі семисегментного індикатора

Кожен сегмент світлодіодного індикатора е звичайним світло діодом. А їх спільне ввімкнення визначає тип індикатора (з спільним анодом чи катодом). Якщо використовується одноцифровий елемент - при підключенні можна обмежитись одним портом для управління сегментами і приєднанням спільного електрода до плюса чи мінуса. В разі використання декількох цифрових елементів - використовують динамічну індикацію. При такому режимі розряди

індикатора працюють не одночасно, а по черзі. Переключення розрядів відбувається з великою швидкістю (50 Гц), через це людське око не помічає, що індикатори працюють по черзі.

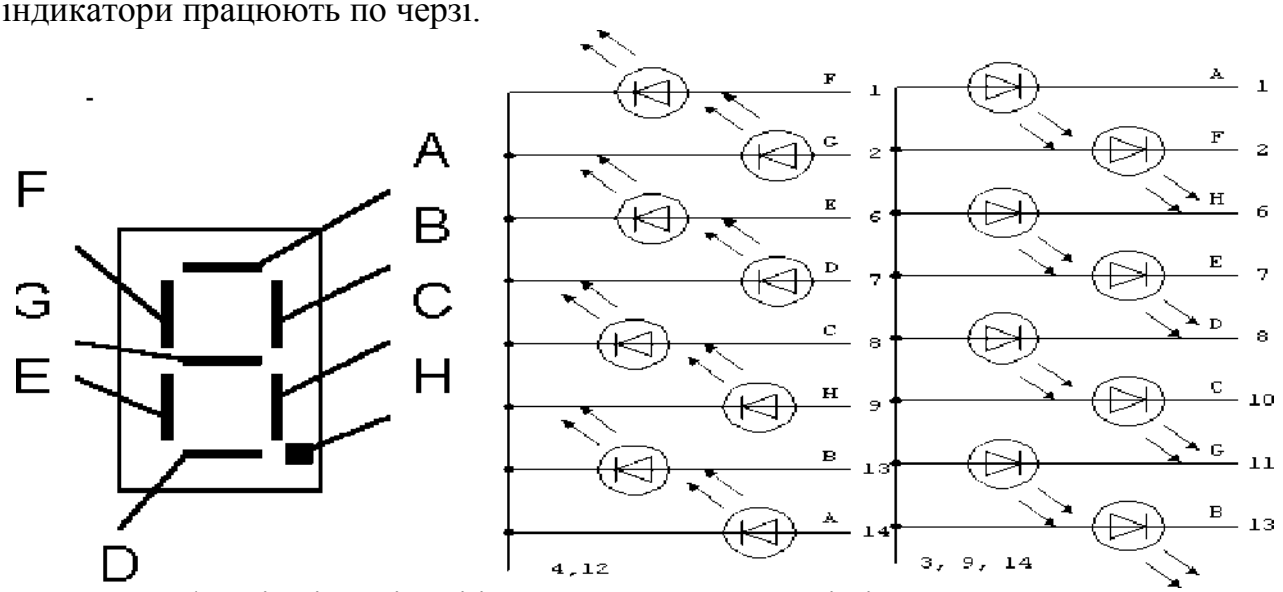


Рис. 2.4. Цифрові світлодіодні індикатори та їх схемні рішення

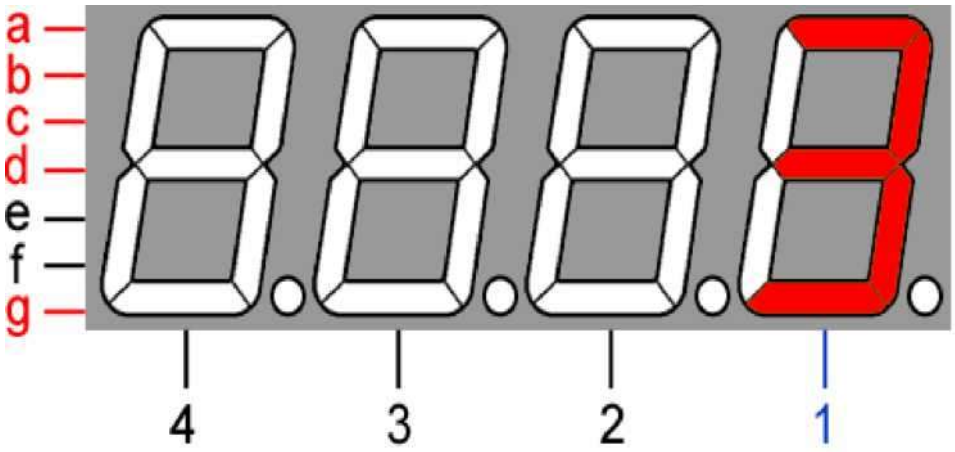


Рис. 2.5. Режим динамічної індикації

Так як у світлодіодів дуже мала інерційність, розряди, що змінюються зливаються в одне зображення. У цьому режимі в кожен момент часу працює тільки один розряд, вмикаються по черзі починаючи з першого закінчуючи останнім, потім все починається спочатку.

Динамічний спосіб відображення інформації базується на тому, що будь-який світловий індикатор є інерційним приладом, а для людського ока зображення на дисплеї , якщо його оновлювати із частотою приблизно 20 разів в секунду, представляється незмінним.

Схема реалізації динамічної індикація без додаткових елементів наведена на рис. 2.6. До порту В МК підключені катоди всіх світлодіодів матриці, а до порту А - аноди кожного з індикаторів, що створюють матрицю. На лініях порту А організовується одиниця, що "біжить". На лінії порту В при кожному положенні одиниці, що біжить, виводиться семисегментний код того символу, який повинен горіти в даному знакомісці. Для індикаторів із загальним катодом замість одиниці,

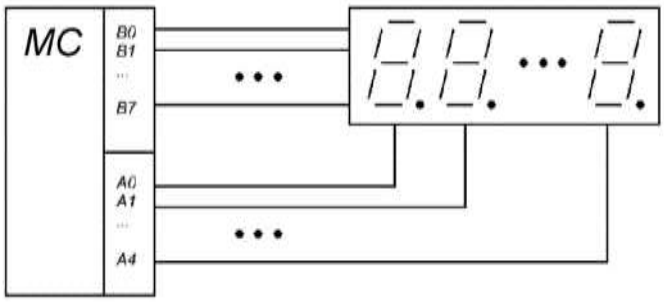


Рис. 2.6. Підключення світлодіодного індикатора без допоміжних елементів

що біжить, використовується нуль, що біжить. Перевага такого способу індикації - у відсутності яких-небудь додаткових компонентів (окрім самих світлодіодних індикаторів), головний недолік - значна перевитрата ліній портів.

.

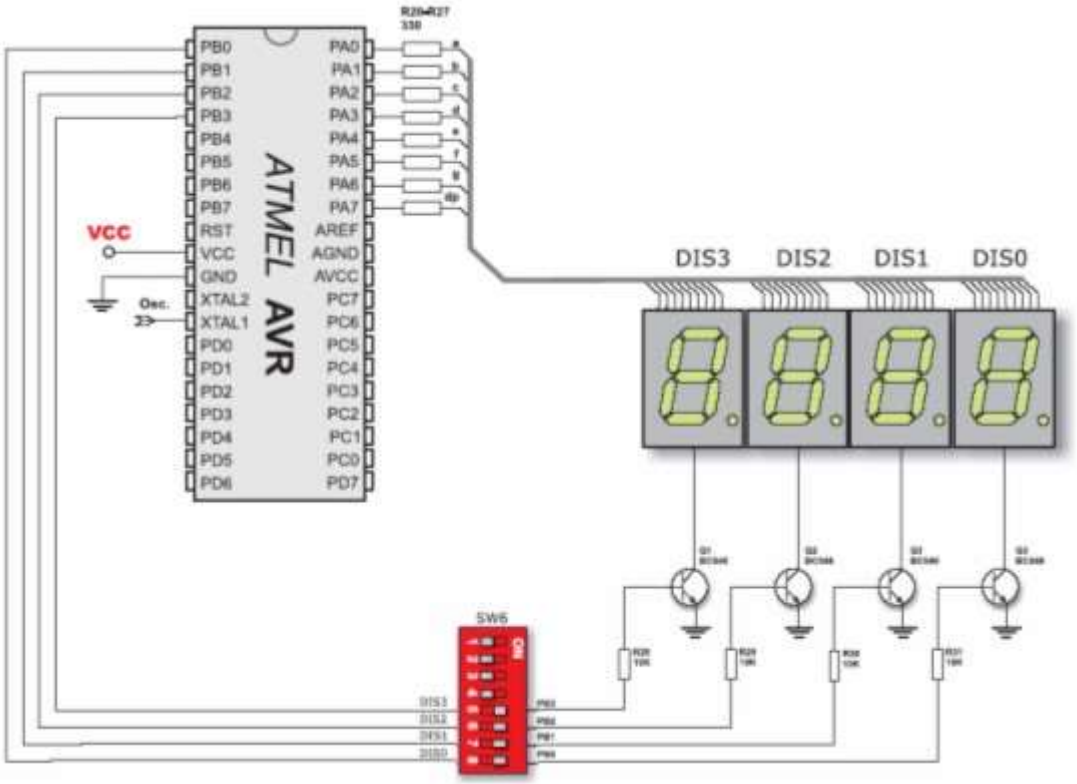


Рис. 2.7. Схема ввімкнення світлодіодного індикатора на навчальній платі

На навчальній платі використовується семи сегментний чотирьох розрядний світлодіодний індикатор з спільним катодом, підключений до порту B. Це значить, що в разі його приєднання за схемою, приведеною на рис. 2.7 - керувати засвічуванням одного елемента ми повинні були низьким рівнем сигналу («0»). Проте на платі використовується підхід з використанням драйвера на транзисторних ключах для управління засвічуванням кожного з елементів. Тобто, для засвічування кожного елемента потрібно використовувати високий рівень сигналу («1»), як у випадку світлодіодного індикатора з спільним анодом

Виведення цифр на світлодіодний індикатор

Наведемо приклад виведення на світлодіодний індикатор номеру кожного з сегментів. (На першому сегменті - цифра «1», на другому - «2», і т.д.)/

```
// A    B C        D E        F G        H(dp)
// B0   B1 B2      B3 B4      B5 B6      B7

//      Digit1     Digit2     Digit3     Digit4

//      D2         D3         D4         D5
```

```
#include <util/delay.h>

#include <avr/io.h>



int main(void)
{
DDRB = DDRD = 0xFF;
PORTB = PORTD = 0xFF;
unsigned char i=20; //затримка мікросекунд

for(;;)
{
        PORTD = 0b00000100; //вмикаємо розряд "1", всі інші - вимикаємо
     PORTB = 0b00000110; // комбінація вмикає цифру "1"
        _delay_us(i);       //часова затримка в мікросекундах на
                            //свічення даного розряду


          PORTD = 0b00001000; //вмикаємо розряд "2", всі інші - вимикаємо
        PORTB = 0b11011011; // комбінація вмикає цифру "2"
        _delay_us(i);

        PORTD = 0b00010000; //вмикаємо розряд "3", всі інші - вимикаємо
        PORTB = 0b01001111; // комбінація вмикає цифру "3"
        _delay_us(i);

        PORTD = 0b00100000; //вмикаємо розряд "4", всі інші - вимикаємо
        PORTB = 0b01100110; // комбінація вмикає цифру "4"
        _delay_us(i);
    }
    return 0;
    }
```

На початку програми усі лінії портів B та D встановлюються як виходи з високим рівнем вихідного сигналу.

В нескінченному циклі відбувається почергове засвічення кожного з розряду індикатора (порт D) з одночасним вмиканням на портові B комбінації пінів, яка створює на екрані ту чи іншу цифру.

Хід роботи

1. Знайти на навчальній платі світлодіодний індикатор.
2. Розібратися з принципом роботи тестової програми та призначенням кожного оператора у програмі.
3. Створити свою власну програму, яка забезпечує функції, описані в індивідуальному завданні, та скомпілювати її.
4. Запрограмувати МК, перевірити правильність виконання.

Індивідуальні завдання

1. Пропонується вивести на світлодіодний індикатор дату свого народження в форматі «день народження, крапка, місяць народження».

Контрольні запитання

1. Назвати існуючі типи семи сегментних світлодіодних індикаторів.
2. Розкрити принцип динамічної індикації.
3. Як кодується зображення довільного символу на семисегиентному світлодіодному індикаторі?
4. Як розраховується час свічення кожного сегменту для динамічного методі відображення інформації?
5. Схеми ввімкнення одиничних семисегментних світлодіодних індикаторів.

Зміст звіту

1. Тема та мета роботи.
2. Перелік використаного обладнання.
3. Стислий зміст теоретичних відомостей.
4. Лістинг власної програми з детальним поясненням кожного рядка.
5. Відповіді на контрольні запитання.
6. Висновки.

Лабораторна робота № 4 ВИКОРИСТАННЯ КНОПОК ДЛЯ КЕРУВАННЯ РОБОТОЮ МІКРОКОНТРОЛЕРА

Мета роботи: ознайомитись з портами вводу-виводу мікроконтролера, принципом обробки сигналів дискретних датчиків. Набути навичок відображення інформації за допомогою світлодіодних індикаторів.

Обладнання: навчально-відлагоджувальна плата AVR-Easy; мікроконтролери ATtiny2313, ATmega8, ATmega16, Atmega8515; середовища програмування WinAVR, Codevision AVR, Flowcode for AVR; внутрішньосхемний програматор.

Теоретичний матеріал

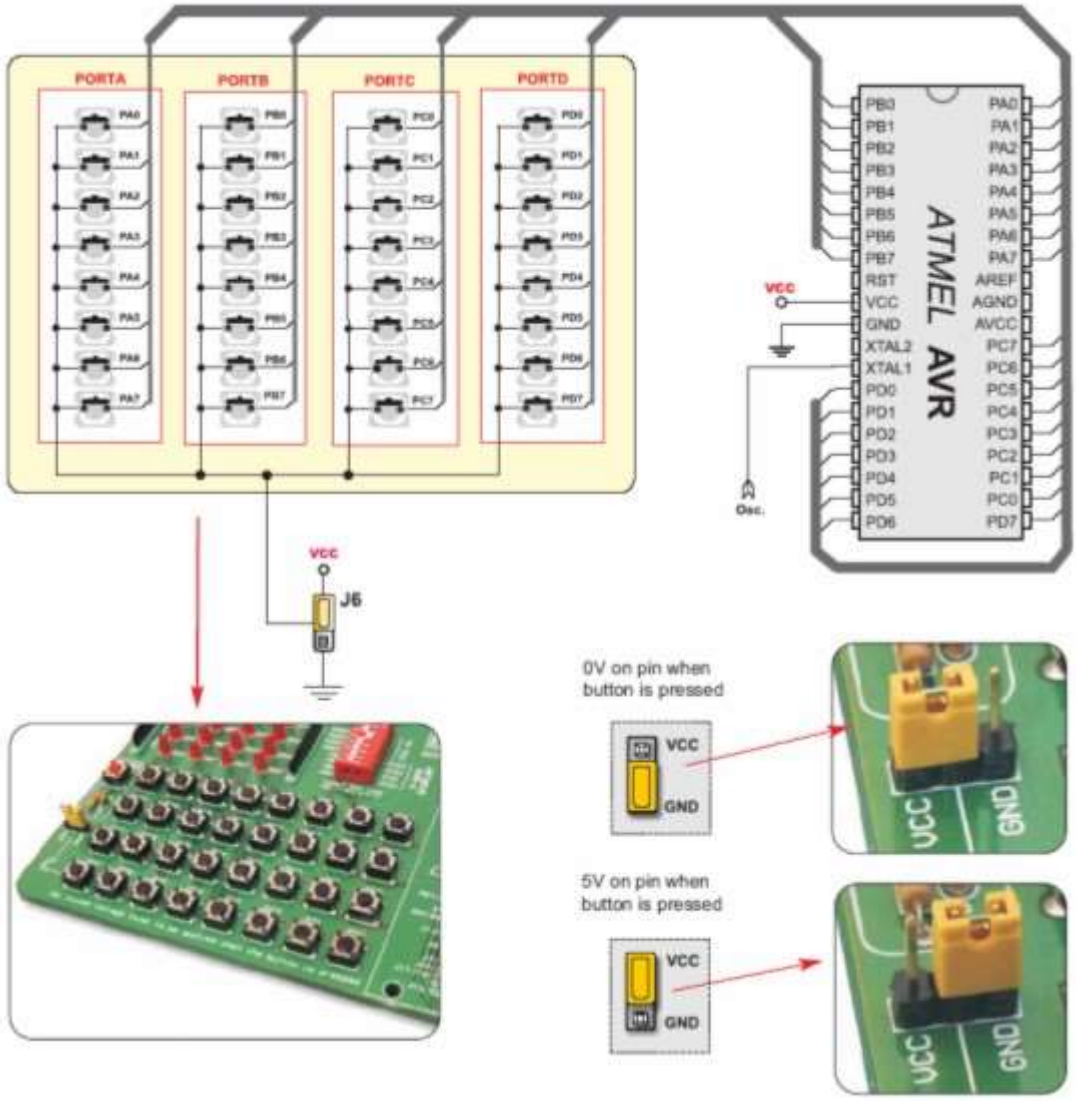


Рис. 2.8. Схема підключення кнопок до мікроконтролера на навчально - відлагоджувальній платі AVR-Easy-Kit.

Навчальна відлагоджувальна плата містить кнопку початкової установки контролера RESET і 32 кнопки, що дозволяють симулювати вхідні дії на порти A, B, C,D. Загальний вивід кнопок може бути підключений до ланцюга живлення або "землі" за допомогою J6. Це дає змогу посилати в порти мікроконтролера сигнали високого або низького рівня.

Управління світлодіодами за допомогою кнопок

Якщо натиснута кнопка D0, горить два з лінійки світлодіодів порту B, а якщо ні то всі світлодіоди. J6 підключено до 5 В.

```
#define F_CPU 8000000UL  // частота в герцах

  #include <avr/io.h>
  #include <util/delay.h>

  int main(void) {

  DDRB = 0xff;              // всі виводи порту B сконфігурувати как виходи
  DDRD = 0x00;        // всі виводи порту D сконфігурувати как входи
  PORTB = 0xff;        // подати всі виводи порту B логічні одиниці
  PORTD = 0x00;  //  виводи порту D  в високоімпедансний стан
start:                    //

  if ((PIND&0x01)==1) // перевірка чи натиснута клавіша D0
  {// зчитуємо стан порта D

  _delay_ms(50);   // затримка для усунення брязкоту контактів
   if ((PIND&0x01)==1) // перевірка чи ще натиснута клавіша D0
   PORTB = 0x88; // запалення двох свытлодіодів
        }
  else
 PORTB = 0xff;

goto start;


  }
```

На початку програми усі лінії порту B встановлюються як виходи з високим рівнем. Усі лінії порту D встановлюються як входи з високоімпедансним станом.

Управління індикаторами за допомогою кнопок

Якщо натиснута кнопка D0, на лівій частині індикатора горить число 99, інакше - 11. Якщо натиснута кнопка D1- то на правій частині горить 87, інакше - 23. Кожна кнопка управляє своєю половиною індикатора незалежно одна від одної. Завжди горять перша і четверта десяткові крапки.

```
// A    B   C   D   E   F   G  H(dp)
// A0 A1 A2 A3 A4 A5 A6 A7

// Digit 1          Digit2          Digit3          Digit4
// B0                  B1              B2              B3

#include <avr/io.h>
```

```
#include <util/delay.h>

int main(void)
{
DDRA = DDRB = 0xFF; // всі розряди портів D і A на вихід
DDRD = 0b00000000; // всі розряди порта D на вхід
PORTB = 0xFF;
PORTD = 0xFF;// включення на D внутрішніх підтягуючих резисторів до 1

unsigned char i=20; //затримка мікросекунд

for(;;)
{
PORTB |= _BV(0);
PORTB &= ~_BV(1);
PORTB &= ~_BV(2);
PORTB &= ~_BV(3); //вмикаємо розряд "1", всі інші – вимикаємо
        // визначення чи нульовий біт D дорівнює нулю
        if(bit_is_clear(PIND, PD0)) PORTA = 0b00000110;
        // комбінація вмикає цифру "1"
        else PORTA = 0b01101111; // комбінація вмикає цифру "9"
        _delay_us(i);
        //часова затримка в мікросекундах на свічення даного розряду
        PORTD &= ~_BV(0);
        PORTD |= _BV(1);
        PORTD &= ~_BV(2);
        PORTD &= ~_BV(3); //вмикаємо розряд "2", всі інші – вимикаємо
        // визначення чи нульовий біт D дорівнює нулю
        if(bit_is_clear(PIND, PD0)) PORTA = 0b00000110;
        // комбінація вмикає цифру "1"
        else PORTA = 0b01101111; // комбінація вмикає цифру "9"
        _delay_us(i);

        PORTD &= ~_BV(0);
        PORTD &= ~_BV(1);
        PORTD |= _BV(2);
        PORTD &= ~_BV(3);
        //вмикаємо розряд "3", всі інші - вимикаємо
        if(bit_is_clear(PIND, PD1)) PORTA = 0b01011011;
        // комбінація вмикає цифру "2"
        else PORTA = 0b01111111; // комбінація вмикає цифру "8"
        _delay_us(i);

        PORTD &= ~_BV(0);
```

```
            PORTD &= ~_BV(1);
            PORTD &= ~_BV(2);
             PORTD |= _BV(3); //вмикаємо розряд "4", всі інші - вимикаємо
            if(bit_is_clear(PIND, PD1)) {PORTA = 0b01001111;}
            // комбінація вмикає цифру "3"
            else PORTA = 0b00000111; // комбінація вмикає цифру "7"
            _delay_us(i);
    }
    return 0;
    }
```

Хід роботи

1. Знайти на навчальній платі потрібні кнопки та світлодіодні індикатори.
2. Розібратися з принципом роботи тестових програм та призначенням кожного оператора у програмі.
3. Створити свою власну програму, яка забезпечує функції, описані в індивідуальному завданні, та скомпілювати її.
4. Запрограмувати МК, перевірити правильність виконання.

Індивідуальні завдання

1. На індикаторі світиться число 0. Коли натискають будь-яку кнопку, число збільшується на 1111 (вийдуть числа 1111, 2222 тощо) з інтервалом 300 мс. Після досягнення числа 9999 збільшення припиняється і програма повертається у початковий стан.
2. На індикаторі світиться число 1234. Коли натискають першу кнопку, число збільшується на 100, коли другу — на 1.
3. На індикаторі світиться число 9999. При натисненні першої кнопки остання цифра зменшується на 1 (виходить 9998), при наступному натисненні — третя цифра зменшується на 1 (виходить 9988), аналогічно для другої і першої цифри. Потім зменшується знову остання цифра.
4. Коли не натиснута жодна кнопка, світиться число 1111, коли натиснута перша — 8811, друга — 1188, обидві — 8888.
5. На індикаторі світиться число 4444. Коли натискають першу кнопку, число зменшується вдвічі, коли другу — збільшується на 123.
6. На індикаторі світиться число 9999. Коли натискають будь-яку кнопку, число зменшується на 1111 (вийдуть числа 8888, 7777 тощо) з інтервалом 300 мс. Після досягнення числа 0000 зменшення припиняється і програма повертається у початковий стан.
7. Коли не натиснута жодна кнопка, світиться число 9999, коли натиснута перша — 4949, друга — 9494, обидві — 4444.
8. На індикаторі світиться число 4320. Коли натискають будь-яку кнопку, число збільшується на 40 (вийдуть числа 4360, 4400 тощо) з інтервалом

300 мс. Після досягнення числа, більшого за 5000, збільшення припиняється і програма повертається у початковий стан.

9. На індикаторі світиться число 8675. Коли натискають будь-яку кнопку, число зменшується на 85 (вийдуть числа 8590, 8505 тощо) з інтервалом 300 мс. Після досягнення числа, меншого за 7500, зменшення припиняється і програма повертається у початковий стан.

10. На індикаторі світиться число 1. Коли натискають будь-яку кнопку, число збільшується вдвічі (вийдуть числа 2, 4, 8 тощо) з інтервалом 300 мс. Після досягнення числа, більшого за 9999, збільшення припиняється і програма повертається у початковий стан.

11. На індикаторі світиться число 0. При першому натисненні першої кнопки перша цифра збільшується на 1 (виходить 1000), при другому натисненні — друга цифра збільшується на 1 (виходить 1100), аналогічно для третьої і четвертої цифри. Потім збільшується знову перша цифра.

12. На індикаторі світиться число 1. Коли натискають будь-яку кнопку, число збільшується втричі (вийдуть числа 3, 9, 27 тощо) з інтервалом 300 мс. Після досягнення числа, більшого за 9999, збільшення припиняється і програма повертається у початковий стан.

13. На індикаторі число збільшується на 1 з інтервалом 1 мс. При натисненні першої кнопки збільшення припиняється та число тримається на індикаторі 5 секунд. Потім програма повертається у початковий стан. Рахунок продовжується з нуля.

14. На індикаторі світиться число 5678. Коли натискають першу кнопку, число зменшується на 200, коли другу — на 2.

15. Коли не натиснута жодна кнопка, світиться число 6543, коли натиснута перша — 1234, друга — 2198, обидві — 3333.

Контрольні запитання

1. Привести основні можливі схемні рішення для приєднання кнопок до мікроконтролерів.
2. Для кожного з приведених схемних рішень включення кнопок привести приклади попередньої ініціалізації порту та подальшого програмного методу опрацювання події натиснення на кнопку.

Зміст звіту

1. Тема та мета роботи.
2. Перелік використаного обладнання.
3. Стислий зміст теоретичних відомостей.
4. Лістинг власної програми з детальним поясненням кожного рядка.
5. Відповіді на контрольні запитання.

Висновки.

Додаток

Робота з портами у мові С

Старе позначення	Можлива заміна	Дія
outp(0xff,DDRB);	DDRB = 0xff;	всі виводи порту В конфігурувати як виходи
outp(0xff,PORTB);	PORTB = 0xff;	у всіх битах порту В встановити "1"
sbi(DDRB, DDB2);	DDRB |= 1<<2;	конфігурувати лінію 2 порту В як вихід
cbi(DDRB, DDB2);	DDRB &= ~(1<<2);	конфігурувати лінію 2 порту В як вхід
sbi(PORTB,PB2);	PORTB |= _BV(PB2); або PORTB |= 1<<2; або PORTB |= 1<<PINB2;	встановити "1" на лінії 2 порту В
cbi(PORTB,PB2);	PORTB &= ~_BV(PB2); або PORTB &= ~(1<<2); або PORTB &= ~1<<PINB2;	встановити "0" на лінії 2порту В

if (bit_is_set(PIND,3))
{
...
}
	if (PIND & (1<<PIND3))
	{
	...
	}
		перевірити "1" на лінії 3 порту D

if (bit_is_clear(PIND,3))
{
...
}
	if (!(PIND & (1<<PIND3)))
	{
	...
	}
		перевірити "0" на лінії 3 порту D D

Зверніть увагу: використання _BV () більш доцільно, так як в цьому випадку компілятор сам виконує порозрядному зрушення і вставляє результат в компільований код. Це забезпечує відсутність витрат часу під час безпосереднього виконання коду в мікроконтролері.

Лабораторна робота № 5 РІДКОКРИСТАЛІЧНИЙ ІНДИКАТОР

Мета роботи: ознайомитись з базовими типами мікроконтролерів AVR. Набути навиків роботи з рідкокристалічними індикаторами, способів виведення текстової інформації на табло, робота з рядками.

Обладнання: навчально-відлагоджувальна плата AVR-Easy; мікроконтролери ATmega16; середовища програмування AVR Studio 4.19; внутрішньосхемний програматор.

Теоретичний матеріал

Алфавітно-цифрові рідкокристалічні індикатори

У даній роботі розглядаються алфавітно-цифрові рідкокристалічні індикатори (LCD) на основі контролера, сумісного з HD44780 (WH1602 від Winstar'a). Вони відображають символи ASCII з кодами від 32 до 122 та деякі інші, залежно від виробника та моделі. Також можна запрограмувати свої символи.

На навчальній платі встановлено індикатор розміру 16x2. LCD під'єднаний по 4-бітному інтерфейсу до порту A.

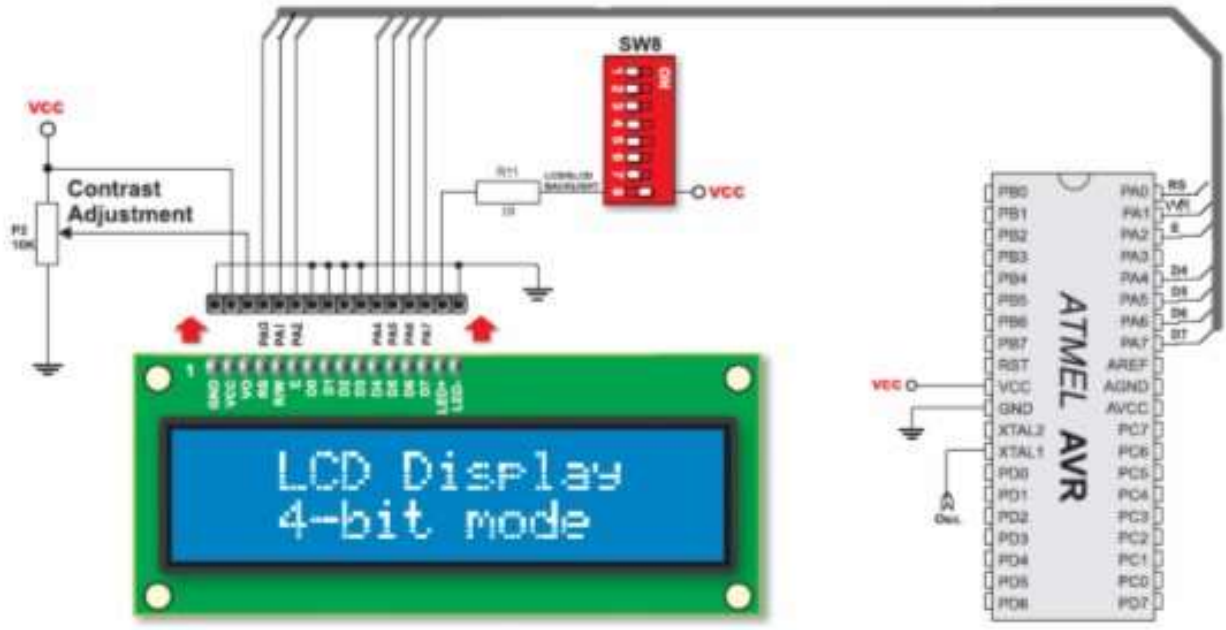


Рис. 2.9. Схема підключення алфавітно-цифрового рідкокристалічного індикатора до мікроконтролера на навчально-відлагоджувальній платі AVR- Easy-Kit

Входи/виходи дисплея

Керуючі входи

RS - Register Select За допомогою цього виводу ми повідомляємо дисплею який тип даних буде записаний / прочитаний. RS = 0 , працюємо з регістром команд (Instruction Register) , RS = 1 , працюємо з регістром даних (Data Register)

RW - Read / WriteЗа допомогою цього висновку, ми перемикаємо дисплей в режим запису або читання. RW = 0 , записуємо дані / команди в дісплей RW = 1 , читаємо дані / змінні з дісплей

E - Enable За допомогою цього висновку, активізується виконання операції запису / читання команд / даних. Або іншими словами, на цей висновок подається "стробірующій сигнал", без якого не може бути виконана жодна операція.Виконання операцій дисплеєм, починаються при спадающем фронті

У більшості випадків читати дані / параметри не приходиться - так що висновок RW можна сміливо підключати до Vss (земля), тобто дисплей весь час буде працювати в режимі запису (Write mode).

Шина даних / адрес

DB7 .. DB0 - Data Bus Символьний дисплей може працювати як з 8-бітної шиною даних / адрес, так і з 4-бітної шиною даних / адрес - що дозволяє заощадити дорогоцінні висновки мікроконтролера. Шина підключається безпосередньо до мікроконтролера, без жодних додаткових перетворень логічних рівнів, вона толерантна як 5В так і до 3.3В -. DB7 - найбільш значущий біт. DB0 - найменш значущий біт.

Якщо використовується 4-бітна шина даних, то в цьому випадку використовуються останні (старші) чотири біта:DB4 .. DB7 , а перші чотири підключаються до землі.

Пам'ять індикатора ділиться на три складові частини:DDR RAM (Display Data RAM), призначена для зберігання 8-бітних символів (в основному ASCII), які ми хочемо відображати на екрані.

Ємність цієї пам'яті становить 80 символів, по 40 на кожен рядок. Решта символи приховані.

Щоб їх відобразити, слід призначити іншу ділянку DDRAM пам'яті (2 × 16 символів) як видиму, за замовчуванням видима пам'ять починається з адреси 0 × 00.

CGROM (Character Generator ROM), тут зберігається розшифровка записаних в DDRAM символів. Тобто, коли ми записуємо в DDRAM комірку, скажімо, символ 0 × 41, то на екрані з'явиться символ зберігається в комірці CGROM пам'яті з адресою 0 × 41 - "A". Як і ASCII таблиці для різних країн, символи зберігаються в CGROM пам'яті відрізняються, так що вибирайте індикатор з потрібною вам ASCII таблицею (CGROM пам'яттю).

CGRAM (Character Generator RAM), в загальному це маленький ділянка CGROM-пам'яті в якій немає ніяких символом і яку можна змінювати - перші 64 байта CGROM пам'яті. Так що користувач може намалювати свої символи. Для того щоб пам'ятати адресу останньої клітинки, до якої ми зверталися, є спеціальний регістр - address counter, за замовчуванням він вказує на комірку 0 × 00, DDRAM пам'яті. Після кожного звернення до пам'яті він автоінкрементіруется або декрементируется в залежності від налаштувань режиму введення.системи команд.

Дисплей розпізнає всього 11 команд, в які входять і команди ініціалізації

дисплея.

Бібліотека для роботи з РК-дисплеєм

Для зручності написання програм з використанням РК індикаторі - доречно винести всі команди і процедури для роботи з індикатором в окрему бібліотеку. Для функціонування даної бібліотеки потрібно щоб в основній програмі була підключена бібліотека затримок “util/delay.h”

Особливості

- Працює з компіляторами IAR AVR, CodeVision AVR, GNU GCC,
- Підтримує lcd контролери HD44780 і KS0066,
- Підтримує підключення lcd до довільних виводів мікроконтролера,
- Підтримує 4-х і 8-ми розрядний інтерфейс,
- Має функції виведення рядків з ОЗУ і флеш,
- Має функції додавання призначених для користувача символів.

Склад бібліотеки

compilers_4.h - файл для підтримки трьох компіляторів

port_macros.h - макроси віртуальних портів

lcd_lib_2.h - заголовки LCD бібліотеки з прототипами функцій і настройками

lcd_lib_2.c - файл реалізації функцій LCD бібліотеки

Підключення до проекту

1. Переписуємо всі файли бібліотеки в папку проекту.
2. Підключаємо lcd_lib_2.c до проекту всередині середовища розробки.
3. Вставляємо заголовок lcd_lib_2.h до Сі файлу, в якому будуть використовуватися lcd функції.
4. Налаштовуємо конфігурацію lcd в заголовки lcd_lib_2.h
5. Прописуємо в код виклик функцій lcd бібліотеки.

Налаштування конфігурації

Налаштування конфігурації у файлі lcd_lib_2.h включає в себе наступні кроки.

1. Налаштування віртуального або реального порту, до якого підключається LCD

Синтаксис оголошення віртуального порту докладно описаний у файлі port_macros.h. У заголовки lcd_lib_2.h вже оголошений порт, в цих оголошення потрібно міняти тільки букви порту (A, B, C ..), номери виводів(0, 1, 2, 3 ...), тип порту (_REAL, _VIRT) , активний рівень (_HI, _NONE). Все інше (ім'я порту та імена висновків) чіпати не треба.

Приклад оголошення віртуального порту для 8-ми бітної шини і реального порту для 4-х бітної шини.

```
// виртуальный порт

//шина данных LCD
#define LCD_PORT  LCD_DATA, F, _VIRT

#define LCD_DATA_0  D, 0, _HI
#define LCD_DATA_1  D, 1, _HI
#define LCD_DATA_2  D, 2, _HI
#define LCD_DATA_3  C, 5, _HI
#define LCD_DATA_4  C, 6, _HI
#define LCD_DATA_5  C, 7, _HI
#define LCD_DATA_6  B, 4, _HI
#define LCD_DATA_7  B, 5, _HI

//управляющие выводы LCD
#define LCD_RS  C, 0, _HI
#define LCD_RW  C, 1, _HI
#define LCD_EN  C, 2, _HI
```

```
// реальный порт

//шина данных LCD
#define LCD_PORT  LCD_DATA, A, _REAL

#define LCD_DATA_0  A, 0, _NONE
#define LCD_DATA_1  A, 1, _NONE
#define LCD_DATA_2  A, 2, _NONE
#define LCD_DATA_3  A, 3, _NONE
#define LCD_DATA_4  A, 4, _HI
#define LCD_DATA_5  A, 5, _HI
#define LCD_DATA_6  A, 6, _HI
#define LCD_DATA_7  A, 7, _HI

//управляющие выводы LCD
#define LCD_RS  A, 0, _HI
#define LCD_RW  A, 1, _HI
#define LCD_EN  A, 2, _HI
```

Рамками виділені ті частини коду, які потрібно налаштовувати під свій проект.

2. Глобальні налаштування драйвера

LCD_CHECK_FL_BF - перевіряти прапор BF або використовувати програмну затримку. 0 - затримка, 1 - перевірка прапора.

LCD_BUS_4_8_BIT - використовувана шина даних. 0 - 4 розрядна шина, 1 - 8-ми розрядна

3. Налаштування ініціалізації дисплея

Ці установки визначають стан дисплей після виклику функції LCD_Init ().

LCD_ONE_TWO_LINE - кількість відображуваних рядків. 0 - 1 рядок; 1 - 2 рядки.

LCD_FONT58_FONT511 - тип шрифту. 0 - 5x8 точок; 1 - 5x11 точок.

LCD_DEC_INC_DDRAM - зміни адреси ОЗУ при виведенні на дисплей. 0 - курсор рухається вліво, адреса зменшується на 1 (текст виходить задом наперед); 1 - курсор рухається вправо, адреса збільшується на 1.

LCD_SHIFT_RIGHT_LEFT - зсув всього дисплея. 0 - при читанні ОЗУ зрушення не виконується, 1 - під час запису в ОЗУ зрушення дисплея виконується згідно з настановою LCD_DEC_INC_DDRAM (0 - зсув вправо, 1 - зсув вліво)

LCD_DISPLAY_OFF_ON - включення / вимикання дисплея. 0 - дисплей вимкнений, але дані в ОЗУ залишаються; 1 - дисплей включений.

LCD_CURSOR_OFF_ON - відображення підкреслює курсору. 0 - курсор не відображається, 1 - курсор відображається.

LCD_CURSOR_BLINK_OFF_ON - відображення миготливого курсору. 0 - миготливий курсор не відображається; 1 - миготливий курсор відображається.

LCD_CURSOR_DISPLAY_SHIFT - команда зсуву вправо / вліво курсора або дисплея без запису на дисплей.

Призначені для користувача макроси і функції

LCD_Clear () - очищення дисплея.

LCD_ReturnHome () - повернення курсору в початкове положення.

LCD_Goto (x, y) - позиціонування курсору. x - номер знакоместа, y - номер рядка.

void LCD_Init (void) - ініціалізація дисплея.

void LCD_WriteCom (uint8_t data) - запис команди

void LCD_WriteData (char data) - вивід одного символу

void LCD_SendStr (char * str) - вивід рядка з ОЗП.

void LCD_SendStrFl (char __flash * str) - вивід рядка з флеш пам'яті.

void LCD_SetUserChar (uint8_t __flash * sym, uint8_t adr) - завантаження споживацького символу в ОЗУ дисплея..

```
#include <util/delay.h>
#include <avr/io.h>
#include "lcd_lib_2.h"
char text Compiler [] = "xz";
__flash uint8_t quarter Note [] = {4,4,4,4,4,4,28,28};
int main (void)
{// Инициализируем дисплей
LCD_Init (); // Завантажуємо користувальницький символ в нульову комірку ОЗП дисплея
LCD_SetUserChar (quarterNote, 0);
// Встановлюємо курсор в 8-ме знакомісце
LCD_Goto (8,0);
// Виводимо рядок на дисплей
LCD_SendStr (textCompiler);
while (1);
return 1;
}.
```

Виведення тексту на LCD

Дана програма ініціалізує LCD і виводить текст “Hello, LCD!!! 16 chars, 2 lines.”

```
#include <util/delay.h>
#include <avr/io.h>
#include "lcd_lib_2.h"
{
DDRB = PORTB = 0xFF;
unsigned char text[] = "—Hello, LCD!!=-16 chars,2 lines.";
LCD_Init ();  //Ініціалізація РК-екрану

LCD_WriteCom (0x80);

//Переведення курсору на початок першого рядка

 for(unsigned char i=0; i<32; i++)
{ if(i == 16)
```

```
        LCD_WriteCom (0xC0);
//Переведення курсору на другий рядок, якщо текст виходить за межі першого
        LCD_WriteData (text[i]); //Виведеня і-го символу з масиву

        }


        for(;;);

        }
```

У нескінченному циклі процесор нічого не робить, але вилучити цикл не можна, щоб МК не скидався і не починав виконувати усю програму спочатку.

В кінці функції main() обов'язково повинен бути нескінченний цикл.

Символи кирилиці

Записувати символи як рядок у програмі дуже зручно, проте це працює лише для символів з кодами ASCII від 32 до 122. Решта символів, зокрема, букви кирилиці, закодовані не так, як у комп'ютері. Щоб правильно вивести їх на екран LCD, необхідна програма, що перекодовує символи. Її можна написати самому. Результатом програми є масив, який необхідно скопіювати у свою програму.

Рядок, що біжить

Програма показує у першому рядку заголовок “-== Планети ==-”, а другий рядок біжить: “Сонячна система містить 8 планет: Меркурій, Венера, Земля, Марс, Юпітер, Сатурн, Уран, Нептун.”.

```
        #include <util/delay.h>
        #include <avr/io.h>
        #include "lcd_lib_2.h"

        unsigned char text[ ] = { '-', 0x3D, 0x3D, 0x20, 0xA8, 0xBB, 0x61, 0xBD,
0x65, 0xBF, 0xB8, 0x20, 0x20, 0x3D, 0x3D, '-' };
        unsigned char text1[] = { 0x20, 0x20, 0x43, 0x6F, 0xBD, 0xC7, 0xC0, 0xBD,
0x61, 0x20, 0x63, 0xB8, 0x63, 0xBF, 0x65, 0xBC, 0x61, 0x20, 0xBC, 0x69, 0x63,
0xBF, 0xB8, 0xBF, 0xC4, 0x20, '8', 0x20, 0xBE, 0xBB, 0x61, 0xBD, 0x65, 0xBF,
0x3A, 0x20, 0x4D, 0x65, 0x70, 0xBA, 0x79, 0x70, 0x69, 0xB9, 0x2C, 0x20, 0x42,
0x65, 0xBD, 0x65, 0x70, 0x61, 0x2C, 0x20, 0xA4, 0x65, 0xBC, 0xBB, 0xC7, 0x2C,
0x20, 0x4D, 0x61, 0x70, 0x63, 0x2C, 0x20, 0xB0, 0xBE, 0x69, 0xBF, 0x65, 0x70,
0x2C, 0x20, 0x43, 0x61, 0xBF, 0x79, 0x70, 0xBD, 0x2C, 0x20, 0xA9, 0x70, 0x61,
0xBD, 0x2C, 0x20, 0x48, 0x65, 0xBE, 0xBF, 0x79, 0xBD,'.' };

        int main(void)
        {
        DDRB = PORTB = 0xFF;
        LCD_Init ();          //Ініціалізація РК-екрану
        LCD_WriteCom (0x80); //Переведення курсору на початок першого рядка
```

```
for(unsigned char i=0; i<16; i++)
        LCD_WriteData (text[i]); //Виведення першого напису
        for(;;)
        { for(unsigned char offset=0; offset<=sizeof(text1)-16; offset++)
           { LCD_WriteCom (0xC0);//Переведення курсору на початок другого рядка
for(unsigned char i=0; i<16; i++) LCD_WriteCom (text1[offset+i]);
           //Виведення 16- ти символів
           for(unsigned char i=0; i<40; i++) _delay_ms(10); //delay 400 ms
           }
        }

        }
```

Оскільки значення першого рядка не змінюються, він передається у LCD один раз на початку програми. У нескінченному циклі передається лише значення другого рядка.

Оператор sizeof(text) визначає довжину масиву text. Якби масив був описаний у вигляді рядка (рядок в Сі — це масив, що закінчується нуль- символом — символом з кодом 0), його довжина виявилась би на одиницю більша, тому що нуль-символ в кінці також враховується в довжину рядка.

Хід роботи

1. Знайти на навчальній платі рідкокристалічний індикатор.
2. Розібратися з принципом роботи тестових програм та призначенням кожного оператора у програмі.
3. Створити свою власну програму, яка робить рядок, що біжить. Текст, який біжить, має складатися з Вашого прізвища та будь-якого речення кирилицею. Назва тексту відображається на першому рядку. Скомпілювати програму.
4. Запрограмувати МК, перевірити правильність виконання.

Контрольні запитання

1. Якими можливостями володіють алфавітно-цифрові рідкокристалічні індикатори? Які ще є рідкокристалічні індикатори?
2. Привести існуючі схемні рішення для підключення алфавітно-цифрових рідкокристалічних індикаторів.
3. Яке призначення кожного з пінів алфавітно-цифрових рідкокристалічних індикаторів.
4. Який алгоритм роботи з алфавітно-цифровим рідкокристалічним індикатором?
5. Пояснити кожну з процедур для роботи з алфавітно-цифровим рідкокристалічним індикатором.

Зміст звіту

1. Тема та мета роботи.

2. Перелік використаного обладнання.
3. Стислий зміст теоретичних відомостей.
4. Лістинг власної програми з детальним поясненням кожного рядка.
5. Відповіді на контрольні запитання.
6. Висновки

Розшифровка команд для LCD

Команда LCD	HEX - код	Виконувані дії	Час виконання., мкс
Очистка дисплею	0x01	Порожній екран, очистка пам'яті, курсор в лівій верхній позиції	1640
Повернення курсору на початок	0x02	Курсор в лівій верхній позиції, пам'ять не очищається	1640
Здвиг курсору вліво	0x04	Після виводу чергового символу курсор автоматично здвигається на одне знакомісце вліво	40
Здвиг курсору вправо	0x06	Після виводу чергового символу курсор автоматично здвигається на одне знакомісце вправо	40
Вимкнення дисплею	0x08	Повна відсутність зображення на екрані LCD	40
Вимкнення курсору	0x0C	Дозволено виведення зображення, проте курсор не видно	40
Прямокутна форма курсору	0x0D	Дозволено виведення зображення, курсор у вигляді блимаю чого чорного прямокутника	40
Лінійна форма курсору	0x0E	Дозволено виведення зображення, курсор у вигляді нижньої підрядкової немигаючої лінії	40
Комплексна форма курсору	0x0F	Дозволено виведення зображення, курсор у вигляді блимаю чого чорного прямокутника з підкресленням	40
Інтерфейс 4 біта, 1 рядок	0x20	Зв'язок з однорядковим LCD через 4 лінії шини даних	40
Інтерфейс 4 біта, 2 рядки	0x28	Зв'язок з дворядковим LCD через 4 лінії шини даних	40
Інтерфейс 8 біт, 1 рядок	0x30	Зв'язок з однорядковим LCD через 8 ліній шини даних	40
Інтерфейс 8 біт, 2 рядки	0x38	Зв'язок з дворядковим LCD через 8 ліній шини даних	40
Доступ до ОЗП знакогенератора	0x40 - 0x7F	Запис даних по цим адресам дозволяє створити 16 власних символів	40
Установка позиції курсору	0x80 - 0xCF	Курсор встановлюється в позицію згідно рис. 2	40

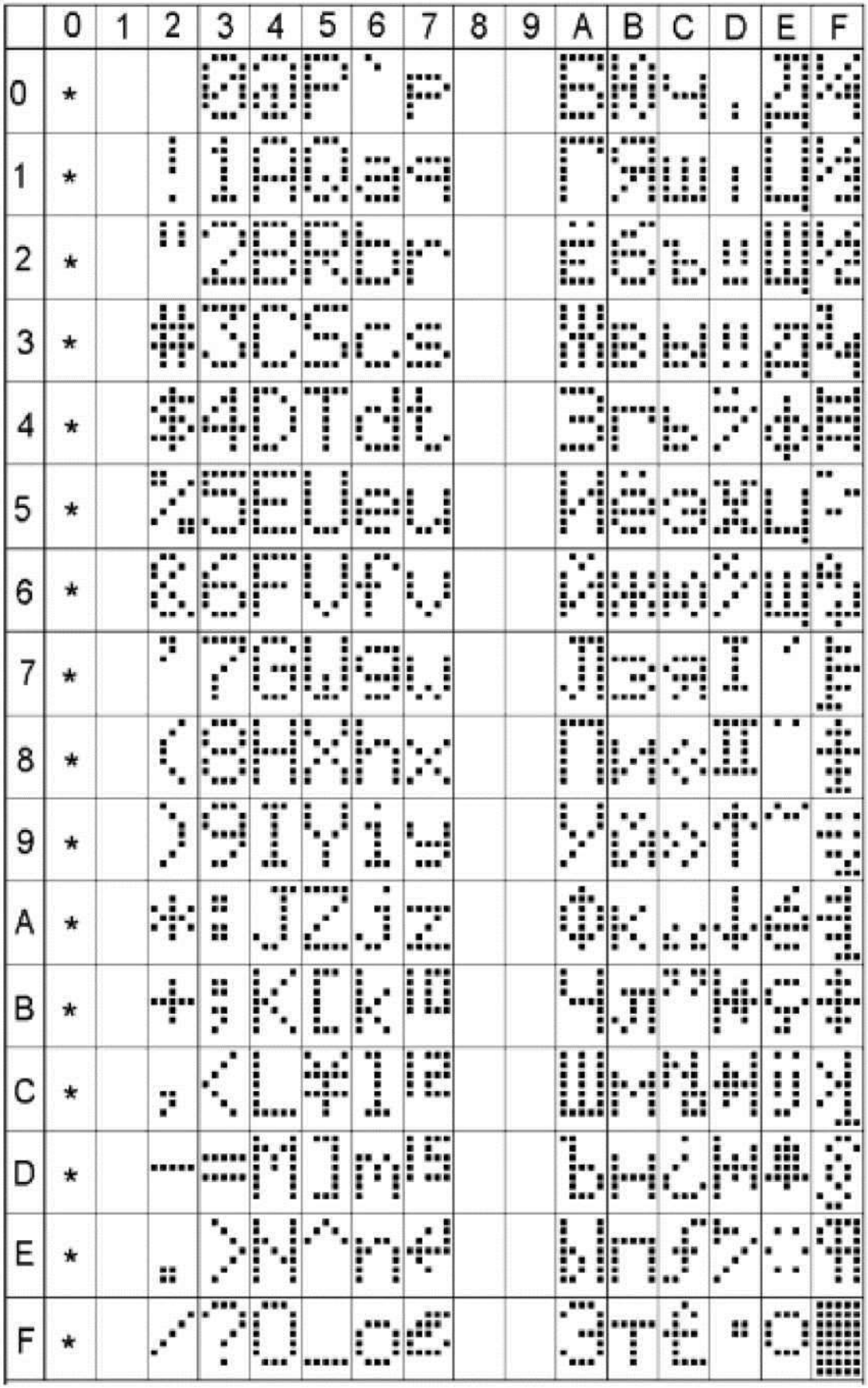
Розподіл адрес на рядках екрану

Верхній рядок LCD

0x80	**0x81**	**0x32**	**0x33**	**0x84**	**0x85**	**0x86**	**0x37**	0x88	**0x89**	**0x8A**	**Ox3B**	**Ox3C**	**0x80**	**0x8 E**	**0x8F**
OxCO	**0xC1**	**0xC2**	**OxC3**	**0xC4**	**0xC5**	**OxCб**	0xC7	**0xC8**	**0xC9**	**OxCA**	**OxCB**	**OxCC**	**OxCD**	**OxCE**	**OxCF**

Нижній рядок LCD

Відповідність символів знакогенератора до адрес в пам’яті ПЗП
ПРИКЛАД: код 0x34 відповідає цифрі “ 4 код 0xB9 - букві “ й ”



Лабораторна робота № 7 АНАЛОГО-ЦИФРОВИЙ ПЕРЕТВОРЮВАЧ В МК AVR

Мета роботи: ознайомитись з базовими типами мікроконтролерів AVR. Набути навиків роботи з вбудованим аналого-цифровим перетворювачем, способів обробки виміряних даних та їх виведення на засоби відображення інформації.

Обладнання: навчально-відлагоджувальна плата AVR-Easy-Kit; мікроконтролери ATmega16; середовище програмування AVR Studio; внутрішньосхемний програматор.

Теоретичний матеріал

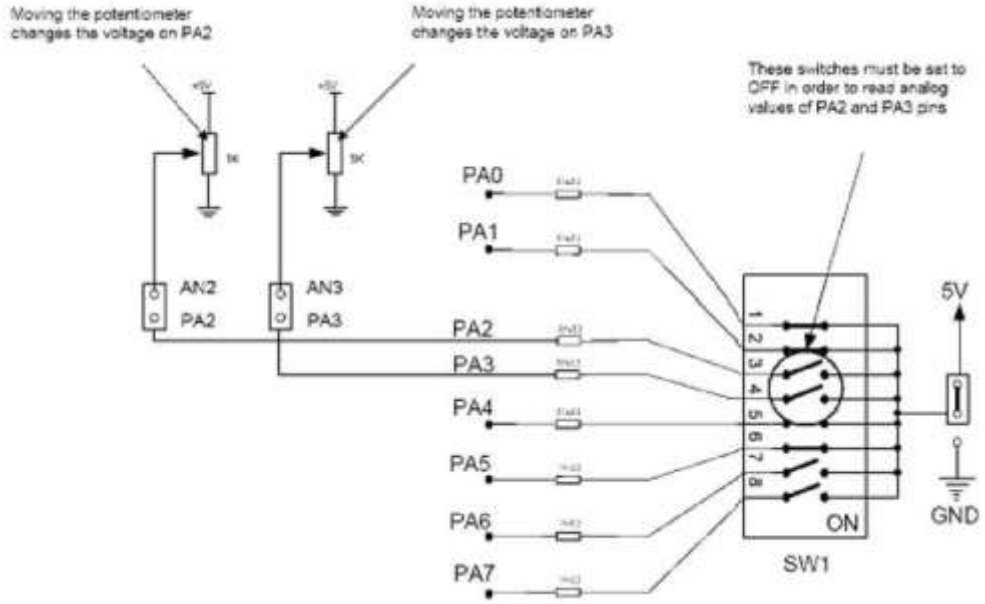


Рис. 1. Схема підключення подільників напруги до каналів АЦП мікроконтролерів

Аналого-цифровий перетворювачмікроконтролерів AVR

Аналого-цифрові перетворювачі (АЦП) є пристроями, які приймають вхідні аналогові сигнали та генерують відповідні їм цифрові сигнали, придатні для обробки мікропроцесорами та іншими цифровими пристроями. Багатоканальний АЦП входить в більшість сучасних моделей МК AVR. Зазвичай число каналів дорівнює 8, але в різних моделях воно може варіювати від 4 каналів в молодших моделях сімейства Tiny, 6 каналів в ATmega8, до 16 каналів в ATmega2560. Багатоканальність означає, що на вході єдиного модуля АЦП встановлений аналоговий мультиплексор, який може підключати цей вхід до різних виводів МК для здійснення вимірювань декількох незалежних аналогових величин з рознесенням по часу. Входи мультиплексора можуть працювати окремо (в несиметричному режимі для виміру напруги відносно "землі") або (в деяких моделях) об'єднуватися в пари для вимірювання диференціальних сигналів. Іноді АЦП додатково забезпечується підсилювачем напруги з фіксованими значеннями коефіцієнта підсилення 10 і 200. Сам АЦП являє собою перетворювач послідовного наближення з пристроєм вибірки- зберігання і фіксованим числом

тактів перетворення, рівним 13 (або 14 для диференціального входу: перше перетворення після ввімкнення потребує 25 тактів для ініціалізації АЦП). Тактова частота формується аналогічно тому, як це робиться для таймерів-за допомогою спеціального дільника тактової частоти МК, який може мати коефіцієнти розподілу від 1 до 128. Але на відміну від таймерів, вибір тактової частоти АЦП не зовсім довільний, так як швидкодія аналогових компонентів обмежена. Тому коефіцієнт ділення слід вибирати таким, щоб при заданій частоті роботи МК тактова частота АЦП укладалася в рекомендований діапазон 50-200 кГц (тобто максимум близько 15 тис. вимірювань в секунду). Збільшення частоти вибірки допустимо, якщо не потрібно досягнення високої точності перетворення. Роздільна здатність АЦП в МК AVR - 10 двійкових розрядів, чого для більшості типових застосувань досить. Абсолютна похибка перетворення залежить від ряду факторів і в ідеальному випадку не перевищує ± 2 молодших розряди, що відповідає загальній точності вимірювання приблизно 8 двійкових розрядів. Для досягнення цього результату необхідно приймати спеціальні заходи: не тільки правильно підбирати тактову частоту в рекомендований діапазон, але і знижувати по максимуму інтенсивність цифрових шумів. Для цього рекомендується, як мінімум, не використовувати невикористані виводи того ж порту, до якого підключений АЦП, для обробки цифрових сигналів, робити правильну розводку друкованої плати, а як максимум - додатково до того ще й включати спеціальний режим ADC Noise Reduction. АЦП може працювати у двох режимах: одиночного і безперервного перетворення. Другий режим доцільний лише при максимальній частоті вибірок. В інших випадках його слід уникати, оскільки обійти в цьому випадку необхідність паралельної обробки цифрових сигналів, як правило, неможливо, а це означає зниження точності перетворення.

Регістри управління АЦП

Для налаштування АЦП існує два регістри: ADCSR (регістр контролю та стану АЦП) та ADMUX (регістр мультиплексора АЦП). У всіх серій МК AVR призначення регістрів та їх бітів налаштування практично не відрізняється.

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
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	

Рис. 2. Розподілення розрядів в регістрі ADCSRA

ATmega8

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Всі інші моделі

Bit	7	6	5	4	3	2	1	0	
	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
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	

Рис. 3. Розподілення розрядів в регістрі ADMUX для різних серій мікроконтролерів AVR

Призначення кожного з бітів вказаних регістрів детально описано в даташитах до кожного мікроконтролера.

В ході роботи з АЦП МК потрібно пам’ятати наступне:

1. Не можна починати нове вимірювання, поки не завершилось попереднє.
2. Вимірювання з декількох каналів не може відбуватись одночасно. В будь-який момент часу можлива робота лише з одною, конкретно вибраною лінією АЦП за допомогою регістру ADMUX. Якщо пристрій працює лише з одним каналом АЦП, то достатньо на початку програми в блокові ініціалізації налаштувати потрібний канал. В противному разі потрібно буде перед кожним вимірюванням пере налаштовувати регістр ADMUX для вибору потрібного каналу.
3. При виборі зовнішнього опорного джерела напруги потрібно щоб відповідні виводи МК були до нього приєднані в схемі (виводи AVCC і AREF).

Вимірювання рівня вхідної напруги на каналі PA2 та виведення значення на LCD

Дана програма ініціалізує LCD, проводить вимірювання вхідного аналогового сигналу на каналі PA2 і виводить на екран в першому рядку назву вибраного каналу, а в другому - виміряне значення рівня сигналу.

```
/*
Target MCU:
ATmega16 Target
device: AVR-
Easy
*/

#include <avr/io.h>
#include <util/delay.h>
```

```
#include "LCD.C"                    //Бібліотека для роботи з РК-екраном
unsigned long u=0; unsigned long voltage=0; unsigned char i;

unsigned char text[] = { '-', 0x3D, 0x3D, 0x20, 'A', 'D', 'C', 0x20 ,'P', 'A', '2',
0x20, 0x3D, 0x3D, '-', 0x20, 0x20 };

//=====Функція зчитування даних з попередньо вибраного каналу АЦП
unsigned int getADC(void)
{ unsigned int v; //локальна змінна
ADCSRA|=(1<<ADSC); //почати претворення
 while ((ADCSRA&_BV(ADIF))==0x00); //Чекаємо закінчення перетворення
v=(ADCL|ADCH<<8); //Зчитуємо значення АЦП
return v;
}
//Головна програма int main(void)
{
DDRB = PORTB = 0xFF; //порт B на вихід, високий рівень
ADMUX = (0<<REFS1)| (1<<REFS0)| (0<<ADLAR)| (0<<MUX4)|
(0<<MUX3)| (0<<MUX2)| (1<<MUX1)| (0<<MUX0);
// ____ AVCC з коденсатором на AREF (REFS1 та REFS0)
 // розрядність10 біт (ADLAR)
//ADC2
ADCSRA = (1<<ADEN)|(1<<ADPS2)|(0<<ADPS1)|(1<<ADPS0);
// ADC in
 //Тактова частота АЦП СК/32
lcd_init( ); //Ініціалізація РК-екрану
lcd_com(0x80); //Переведення курсору на початок першого рядка

for(unsigned char i=0; i<16; i++)

 lcd_dat(text[i]); //Виведення напису

//Безкінечний цикл while(1)
{
u=getADC( );
//Зчитуємо дані з вибраного каналу АЦП
voltage= 5*u*1000/1024; //Розрахунок значення напруги.

//=Виводимо отримане значення на РКІ.
//=Розкладаємо отримане значення на розряди, після першого ставимо кому
 lcd_com(0xC4); //Переводимо курсор на другий рядок на 5-ту позицію
 lcd_dat(voltage/1000+0x30);
 lcd_dat(',');
 i=voltage/1000;
```

```
u=voltage-i*1000;
lcd_dat((u/ 100)+0x30);
i=voltage/100;
u=voltage-i*100;
lcd_dat((u/ 10)+0x30);
lcd_dat(voltage% 10+0x30);
lcd_dat(' ');
lcd_dat('V');
for(unsigned char d = 50; d>0; d--) _delay_ms(10); //Затримка 500 ms
}
}
```

У нескінченному циклі процесор викликає процедуру зчитування даних з вибраного каналу АЦП та виводить обраховане значення на екран.

Хід роботи

1. Знайти на навчальній платі рідкокристалічний індикатор, змінні резистори для регулювання рівня вхідного сигналу на канал АЦП, лінійки світлодіодів та кнопки.
2. Підключити потрібні канали АЦП

.

3. Повторити принципи роботи з рідкокристалічними індикаторами та кнопками. Навчитись налаштовувати вбудований АЦП на різні режими роботи та обробляти виміряні дані.
4. Створити свою власну програму, яка забезпечує функції, описані в індивідуальному завданні, та скомпілювати її.
5. Запрограмувати МК, перевірити правильність виконання.

Індивідуальні завдання

1. Виміряні значення аналогових сигналів з каналів PA2 і PA3 виводяться в центрі РК-екрану. Рівень вхідного сигналу каналу PA2 також відображається на одному з світлодіодних рядків. Кількість лінійно засвічених світлодіодів повинна бути пропорційною до рівня вхідного сигналу.
2. Виміряні значення аналогових сигналів з каналів PA2 і PA3 виводяться в центрі РК-екрану. Рівень вхідного сигналу каналу PA2 або каналу PA3 також відображається на світлодіодному рядку. Передбачити в програмі 1 кнопку, яка б визначала який з каналів відображатиметься на світлодіодному рядку в поточний час. Перемикання каналу відбувається після відпускання кнопки. При цьому в другому рядку повинна відображатись назва вибраного каналу.
3. Виміряні значення аналогових сигналів з каналів PA2 і PA3 виводяться в центрі РК-екрану. Рівень вхідного сигналу каналу PA3 також відображається на нижньому рядку РК-екрану у вигляді рядка з зафарбованих прямокутників. Кількість лінійно зафарбованих прямокутників повинна бути

пропорційною до рівня вхідного сигналу.

4. Виміряні значення аналогових сигналів з каналів PA2 і PA3 виводяться на початку кожного з рядків РК-екрану відповідно. Навпроти кожного з цифрових значень виводиться лінійна шкала відповідного рівня сигналу з чорних прямокутників розмірністю 10. Кількість лінійно зафарбованих прямокутників повинна бути пропорційною до рівня вхідного сигналу.

5. Виміряні значення аналогових сигналів з каналів PA2 і PA3 виводяться в центрі РК-екрану. Рівень вхідного сигналу каналу PA2 або каналу PA3 також відображається на нижньому рядку РК-екрану у вигляді рядка з зафарбованих прямокутників розмірністю 10. Кількість лінійно зафарбованих прямокутників повинна бути пропорційною до рівня вхідного сигналу. Передбачити в програмі 1 кнопку, яка б визначала який з каналів відображатиметься на екрані в поточний момент. Перемикання зканалів відбувається після відпускання кнопки. При цьому на початку другого рядка повинна відображатись назва вибраного каналу.

Контрольні запитання

1. Вказати основні існуючі типи аналого-цифрових перетворювачів та охарактеризувати їх.
2. Характеристики та можливості аналого-цифрового перетворювача в мікроконтролерах AVR
3. Назвати регісти для роботи з АЦП та вказати їх призначення.
4. Що таке опорна напруга та як вона задається для роботи з АЦП?
5. Скільки каналів АЦП існує в МК AVR та як відбувається вимірювання на каналах? Чи можливе одночасне вимірювання на всіх каналах? Якщо ні - то як це подолати?
6. Як програмно задається канал вимірювання аналогового сигналу?
7. Як задається точність вимірювання АЦП?
8. В яких одиницях відбувається вимірювання аналогового сигналу?
9. Які межі вимірювання аналогового сигналу? Як розширити можливі межі вимірювання? Привести приклад схемного рішення.

Зміст звіту

1. Т ема та мета роботи.
2. Перелік використаного обладнання.
3. Стислий зміст теоретичних відомостей.
4. Лістинг власної програми з детальним поясненням кожного рядка.
5. Відповіді на контрольні запитання.
6. Висновки